Б. Керниган, Д. Ритчи



Язык
программирования
Си

Brian W. Kernighan, Dennis M. Ritchie



Second Edition

AT&T Bell Laboratories
Murray Hill, New Jersey

Prentice Hall PTR, Englewood Cliffs, New Jersey 07632

Б. Керниган, Д. Ритчи



Издание 3-е, исправленное

Перевод с английского
под редакцией Вс. С. Штаркмана



Санкт-Петербург 2003

УДК 681.3.06
ББК 24.4.9
К36

Перевод с английского *Вт. С. Штаркмана,*
под редакцией *Вс. С. Штаркмана.*

Керниган Б., Ритчи Д.
К36 **Язык программирования Си.**\Пер. с англ., 3-е изд., испр. — СПб.: "Невский Диалект", 2001. - 352 с: ил.

Книга широко известных авторов, разработчиков языка Си, переработанная и дополненная с учетом стандарта ANSI для языка Си, 2-е английское издание которой вышло в 1988 году, давно стала классикой для всех изучающих и/или использующих как Си, так и Си++. Русский перевод этой книги впервые был выпущен изд-вом "Финансы и статистика" в 1992 г. и с тех пор пользуется неизменным спросом читателей.

Для настоящего третьего русского издания перевод заново сверен с оригиналом, в него внесены некоторые поправки, учитывающие устоявшиеся за прошедшие годы изменения в терминологии, а так же учтены замечания, размещенные автором на странице http://cm.bell-labs.com/cm/cs/cbook/2ediffs.html.

Для программистов, преподавателей и студентов.

Издание подготовлено при участии издательства "Финансы и статистика".

ISBN 5-7940-0045-7
ISBN 0-13-110362-8 {PBK}
ISBN 0-13-110370-9 (англ.)

Оглавление

Предисловие

С момента публикации в 1978 г. книги "Язык программирования Си" в мире компьютеров произошла революция. Большие машины стали еще больше, а возможности персональных ЭВМ теперь сопоставимы с возможностями больших машин десятилетней давности. Язык Си за это время также изменился, хотя и не очень сильно; что же касается сферы применения Си, то она далеко вышла за рамки его начального назначения как инструментального языка операционной системы UNIX.

Рост популярности Си, накапливающиеся с годами изменения, создание компиляторов коллективами разработчиков, ранее не причастных к проектированию языка, - все это послужило стимулом к более точному и отвечающему времени определению языка по сравнению с первым изданием книги. В 1983 г. Американский институт национальных стандартов (American National Standards Institute - ANSI) учредил комитет, перед которым была поставлена цель выработать "однозначное и машинно-независимое определение языка Си", полностью сохранив при этом его стилистику. Результатом работы этого комитета и явился стандарт ANSI языка Си.

Стандарт формализует средства языка, которые в первом издании были только намечены, но не описаны, такие, например, как присваивание структурам и перечисления. Он вводит новый вид описания функций, позволяющий проводить повсеместную проверку согласованности вызовов функций с их определением; специфицирует стандартную библиотеку с широким набором функций ввода-вывода, управления памятью, манипуляций со строками символов и другими функциями; уточняет семантику, бывшую в первоначальном определении неясной, и явно выделяет то, что остается машинно-зависимым.

Во втором издании книги "Язык программирования Си" представлена версия Си, принятая в качестве стандарта ANSI. Мы решили описать язык заново, отметив при этом те места, в которых он претерпел изменения. В большинство параграфов это не привнесло существенных перемен, самые заметные различия касаются новой формы описания и определения функции. Следует отметить, что современные компиляторы уже обеспечили поддержку значительной части стандарта.

Мы попытались сохранить краткость первого издания. Си — небольшой язык, и чтобы его описать большой книги не требуется. В новом издании улучшено описание наиболее важных средств, таких как указатели, которые занимают центральное место в программировании на Си; доработаны старые примеры, а в некоторые главы добавлены новые. Так, для усиления трактовки сложных объявлений в качестве примеров включены программы перевода объявлений в их словесные описания и обратно. Как и раньше, все примеры были протестированы прямо по текстам, написанным в воспринимаемой машиной форме.

Приложение А — это справочное руководство, но отнюдь не стандарт. В нем мы попытались уложить самое существенное на минимуме страниц. По замыслу это приложение должно легко читаться программистом-пользователем; для разработчиков же компилятора определением языка должен служить сам стандарт. В приложении В приведены возможности стандартной библиотеки. Оно также представляет собой справочник для прикладных программистов, но не для разработчиков компиляторов. Приложение С содержит краткий перечень отличий представленной версии языка Си от его начальной версии.

В предисловии к первому изданию мы говорили о том, что "чем больше работаешь с Си, тем он становится удобнее". Это впечатление осталось и после десяти лет работы с ним. Мы надеемся, что данная книга поможет вам изучить Си и успешно его использовать.

Мы в большом долгу перед друзьями, которые помогали нам в выпуске второго издания книги. Джон Бентли, Дуг Гуин, Дуг Макилрой, Питер Нельсон и Роб Пайк сделали четкие замечания почти по каждой странице первого варианта рукописи. Мы благодарны Алу Ахо, Деннису Аллиссону, Джою Кемпбеллу, Г. Р. Эмлину, Карен Фортганг, Аллену Голубу, Эндрю Хьюму, Дэйву Кристолу, Джону Линдерману, Дэйву Проссеру, Гину Спаффорду и Крису Ван Уику за внимательное прочтение книги. Мы получили полезные советы от Билла Чезвика, Марка Кернигана, Эндрю Коэнига, Робина Лейка, Тома Лондона, Джима Ридза, Кловиза Тондо и Питера Вайнбергера. Дейв Проссер ответил на многочисленные вопросы, касающиеся деталей стандарта ANSI. Мы широко пользовались транслятором с Си++ Бьерна Страуструпа для локальной проверки наших программ, а Дейв Кристол предоставил нам ANSI-Си-компилятор для окончательной их проверки. Рич Дрешлер очень помог в наборе книги.

Мы искренне благодарим всех.

Брайан В. Керниган,
Деннис М. Ритчи

Предисловие к первому изданию

Си — это универсальный язык программирования с компактным способом записи выражений, современными механизмами управления структурами данных и богатым набором операторов. Си не является ни языком "очень высокого уровня", ни "большим" языком, не рассчитан он и на какую-то конкретную область применения. Однако благодаря широким возможностям и универсальности для решения многих задач он удобнее и эффективнее, чем предположительно более мощные языки.

Первоначально Си был создан Деннисом Ритчи как инструмент написания операционной системы UNIX для машины PDP-11 и реализован в рамках этой операционной системы. И операционная система, и Си-компилятор, и, по существу, все прикладные программы системы UNIX (включая и те, которые использовались для подготовки текста этой книги[1]) написаны на Си. Фирменные Си-компиляторы существуют и на нескольких машинах других типов, среди которых IBM/370, Honeywell 6000 и Interdata 8/32. Си не привязан к конкретной аппаратуре или системе, однако на нем легко писать программы, которые без каких-либо изменений переносятся на другие машины, где осуществляется его поддержка.

Цель нашей книги — помочь читателю научиться программировать на Си. Издание включает введение-учебник, позволяющий новичкам начать программировать как можно скорее, а также главы, посвященные основным свойствам языка, и справочное руководство. В ее основу положены изучение, написание и проработка примеров, а не простое перечисление правил. Почти все наши примеры — это законченные реальные программы, а не разобщенные фрагменты. Все они были оттестированы на машине точно в том виде, как приводятся в книге. Помимо демонстрации эффективного использования языка, там, где это было возможно, мы стремились проиллюстрировать полезные алгоритмы и принципы хорошего стиля написания программ и их разумного проектирования.

Эта книга не является вводным курсом по программированию. Предполагается, что читатель знаком с такими основными понятиями, как "переменная", "присваивание", "цикл", "функция". Тем не менее и новичок

[1] Имеется в виду оригинал этой книги на английском языке. — *Примеч. пер.*

сможет изучить язык, хотя для него будет очень полезным общение с более знающими специалистами.

Наш опыт показал, что Си — удобный, выразительный и гибкий язык, пригодный для программирования широкого класса задач. Его легко выучить, и чем больше работаешь с Си, тем он становится удобнее. Мы надеемся, что эта книга поможет вам хорошо его освоить.

Вдумчивая критика и предложения многих друзей и коллег помогали нам написать книгу. В частности, Майк Бианки, Джим Блу, Стью Фелдман, Дуг Макилрой, Билл Рум, Боб Розин и Ларри Рослер со вниманием прочли все многочисленные варианты этой книги. Мы в долгу у Ала Ахо, Стива Бьерна, Дана Дворака, Чака Хейли, Марион Харрис, Рика Холта, Стива Джонсона, Джона Машея, Боба Митца, Ральфа Мухи, Питера Нельсона, Эллиота Пинсона, Билла Плейджера, Джерри Спивака, Кена Томпсона и Питера Вайнбергера за полезные советы, полученные от них на различных стадиях подготовки рукописи, а также у Майка Леска и Джо Оссанны за помощь при подготовке ее к изданию.

Брайан В. Керниган,
Деннис М. Ритчи

Введение

Си — универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая как и большинство программ, работающих в ней, написаны на Си. Однако язык не привязан жестко к какой-то одной операционной системе или машине. Хотя он и назван "языком системного программирования", поскольку удобен для написания компиляторов и операционных систем, оказалось, что на нем столь же хорошо писать большие программы другого профиля.

Многие важные идеи Си взяты из языка BCPL, автором которого является Мартин Ричарде. Влияние BCPL на Си было косвенным - через язык B, разработанный Кеном Томпсоном в 1970 г. для первой системы UNIX, реализованной на PDP-7.

BCPL и B - "бестиповые" языки. В отличие от них Си обеспечивает разнообразие типов данных. Базовыми типами являются символы, а также целые и числа с плавающей точкой различных размеров. Кроме того, имеется возможность получать целую иерархию производных типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинно-независимую адресную арифметику.

В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция (`{...}`), ветвление по условию (`if-else`), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, `for`) и с проверкой внизу (do), а также средство прерывания цикла (break).

В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное обращение к себе. Как правило, локальные переменные функции - "автоматические", т. е. они создаются заново при каждом обращении к ней. Определения функций нельзя вкладывать друг в друга, но объявления переменных разрешается строить в блочно-структурной манере. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отно-

шению к функции могут быть внутренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы.

На этапе препроцессирования выполняется макроподстановка в текст программы, включение других исходных файлов и условная компиляция.

Си - язык сравнительно "низкого уровня". Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров, т. е. с символами, числами и адресами. С ними можно оперировать при помощи арифметических и логических операций, выполняемых реальными машинами.

В Си нет прямых операций над составными объектами, такими как строки символов, множества, списки и массивы. В нем нет операций, которые бы манипулировали с целыми массивами или строками символов, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных внутри функций. Нет в нем "кучи" и "сборщика мусора". Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это - механизмы высокого уровня, которые в Си обеспечиваются исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций.

В продолжение сказанного следует отметить, что Си предоставляет средства лишь последовательного управления ходом вычислений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы - и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм.

Отсутствие некоторых из перечисленных средств может показаться серьезным недостатком ("выходит, чтобы сравнить две строки символов, нужно обращаться к функции?"). Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка.

В течение многих лет единственным определением языка Си было первое издание книги "Язык программирования Си". В 1983 г. Институтом американских национальных стандартов (ANSI) учреждается комитет для выработки современного исчерпывающего определения языка Си. Результатом его работы явился стандарт для Си ("ANSI-C"), выпущенный в 1988 г. Большинство положений этого стандарта уже учтено в современных компиляторах.

Стандарт базируется на первоначальном справочном руководстве. По сравнению с последним язык изменился относительно мало. Одной из целей стандарта было обеспечить, чтобы в большинстве случаев существующие программы оставались правильными или вызывали предупреждающие сообщения компиляторов об изменении поведения.

Для большинства программистов самое важное изменение - это новый синтаксис объявления и определения функций. Объявление функции может теперь включать и описание ее аргументов. В соответствии с этим изменился и синтаксис определения функции. Дополнительная информация значительно облегчает компилятору выявление ошибок, связанных с несогласованностью аргументов; по нашему мнению, это очень полезное добавление к языку.

Следует также отметить ряд небольших изменений. В языке узаконены присваивание структур и перечисления, которые уже некоторое время широко используются. Вычисления с плавающей точкой теперь допускаются и с одинарной точностью. Уточнены свойства арифметики, особенно для беззнаковых типов. Усовершенствован препроцессор. Большинство программистов эти изменения затронут очень слабо.

Второй значительный вклад стандарта - это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например чтения-записи файлов), форматного ввода-вывода, динамического выделения памяти, манипуляций со строками символов и т. д. Набор стандартных заголовочных файлов обеспечивает единообразный доступ к объявлениям функций и типов данных. Гарантируется, что программы, использующие эту библиотеку при взаимодействии с операционной системой, будут работать также и на других машинах. Большинство программ, составляющих библиотеку, созданы по образу и подобию "стандартной библиотеки ввода-вывода" системы UNIX. Эта библиотека описана в первом издании книги и широко используется в других системах. И здесь программисты не заметят существенных различий.

Так как типы данных и управляющих структур языка Си поддерживаются командами большинства существующих машин, исполнительная система (run-time library), обеспечивающая независимый запуск и выполнение программ, очень мала. Обращения к библиотечным функциям пишет сам программист (не компилятор), поэтому при желании их можно легко заменить на другие. Почти все программы, написанные на Си, если они не касаются каких-либо скрытых в операционной системе деталей, переносимы на другие машины.

Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Проявляя некоторую дисциплину, можно легко писать переносимые программы,

т. е. программы, которые без каких-либо изменений могут работать на разных машинах. Стандарт предоставляет возможность для явного описания переносимости с помощью набора констант, отражающих характеристики машины, на которой программа будет работать.

Си не является "строго типизированным" языком, но в процессе его развития контроль за типами был усилен. В первой версии Си хоть не одобрялся, но разрешался бесконтрольный обмен указателей и целых, что вызывало большие нарекания, но это уже давным-давно запрещено. Согласно стандарту теперь требуется явное объявление или явное указание преобразования, что уже и реализовано в хороших компиляторах. Новый вид объявления функций - еще один шаг в этом направлении. Компилятор теперь предупреждает о большей части ошибок в типах и автоматически не выполняет преобразования данных несовместимых типов. Однако основной философией Си остается то, что программисты сами знают, что делают; язык лишь требует явного указания об их намерениях.

Си, как и любой другой язык программирования, не свободен от недостатков. Уровень старшинства некоторых операторов не является общепринятым, некоторые синтаксические конструкции могли бы быть лучше. Тем не менее, как оказалось, Си - чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

Книга имеет следующую структуру. Глава 1 представляет собой обзор основных средств языка Си. Ее назначение - побудить читателя по возможности быстрее приступить к программированию, так как мы убеждены, что единственный способ изучить новый язык - это писать на нем программы. Эта часть книги предполагает наличие знаний по основным элементам программирования. Никаких пояснений того, что такое компьютер, компиляция или что означает выражение вида `n = n+1` не дается. Хотя мы и пытались там, где это возможно, показать полезные приемы программирования, эта книга не призвана быть справочником ни по работе со структурами данных, ни по алгоритмам; когда оказывалось необходимым выбрать, на что сделать ударение, мы предпочитали сконцентрировать внимание на языке.

В главах 2-6 различные средства языка обсуждаются более подробно и несколько более формально, чем в главе 1; при этом по-прежнему упор делается на примеры, являющиеся законченными программами, а не изолированными фрагментами. Глава 2 знакомит с базовыми типами данных, с операторами и выражениями. В главе 3 рассматриваются средства управления последовательностью вычислений: `if-else`, `switch`, `while`, `for` и т. д. В главе 4 речь идет о функциях и структуре программы (внешних переменных, правилах видимости, делении программы на несколько исходных файлов и т. д.), а также о препроцессоре. В главе 5 обсуждаются указатели и адресная арифметика. Глава 6 посвящена структурам и объединениям.

В главе 7 описана стандартная библиотека, обеспечивающая общий интерфейс с операционной системой. Эта библиотека узаконена в качестве стандарта ANSI, иначе говоря, она должна быть представлена на всех машинах, где существует Си, благодаря чему программы, использующие ввод-вывод и другие возможности операционной системы, без каких-либо изменений можно переносить с одной машины на другую.

Глава 8 содержит описание интерфейса между программами на Си и операционной системой UNIX, в частности описание ввода-вывода, файловой системы и распределения памяти. Хотя некоторые параграфы этой главы отражают специфику системы UNIX, программисты, пользующиеся другими системами, все же найдут в них много полезных сведений, включая определенный взгляд на то, как реализуется одна из версий стандартной библиотеки, и некоторые предложения по переносимости программ.

Приложение А является справочником по языку. Строгое определение синтаксиса и семантики языка Си содержится в официальном документе стандарта ANSI. Последний, однако, более всего подходит разработчикам компилятора. Наш справочник определяет язык более сжато, не прибегая к педантично юридическому стилю, которым пользуется стандарт. Приложение В - сводка по содержимому стандартной библиотеки и предназначена скорее пользователям, чем реализаторам. В приложении С приводится краткий перечень отличий от первой версии языка. В сомнительных случаях, однако, окончательным судьей по языку остается стандарт и компилятор, которым вы пользуетесь.

Обзор языка

Начнем с быстрого ознакомления с языком Си. Наша цель - показать на реальных программах существенные элементы языка, не вдаваясь в мелкие детали, формальные правила и исключения из них. Поэтому мы не стремимся к полноте и даже точности (заботясь, однако, о корректности примеров). Нам бы хотелось как можно скорее подвести вас к моменту, когда вы сможете писать полезные программы. Чтобы сделать это, мы должны сконцентрировать внимание на основах: переменных и константах, арифметике, управлении последовательностью вычислений, функциях и простейшем вводе-выводе. В настоящей главе мы умышленно не затрагиваем тех средств языка, которые важны при написании больших программ: указателей, структур, большой части богатого набора операторов, некоторых управляющих инструкций и стандартной библиотеки.

Такой подход имеет свои недостатки. Наиболее существенный из них состоит в том, что отдельное характерное свойство языка не описывается полностью в одном месте, и подобная лаконичность при обучении может привести к неправильному восприятию некоторых положений. В силу ограниченного характера подачи материала в примерах не используется вся мощь языка, и потому они не столь кратки и элегантны, как могли бы быть. Мы попытались по возможности смягчить эти эффекты, но считаем необходимым предупредить о них. Другой недостаток заключается в том, что в последующих главах какие-то моменты нам придется повторить. Мы надеемся, что польза от повторений превысит вызываемое ими раздражение.

В любом случае опытный программист должен суметь экстраполировать материал данной главы на свои программистские нужды. Новичкам же рекомендуем дополнить ее чтение написанием собственных маленьких программ. И те и другие наши читатели могут рассматривать эту главу как “каркас”, на который далее, начиная с главы 2, будут "навешиваться" элементы языка.

1.1. Начнем, пожалуй

Единственный способ выучить новый язык программирования - это писать на нем программы. При изучении любого языка первой, как правило, предлагают написать приблизительно следующую программу:

Напечататьслова здравствуй, мир

Вот первое препятствие, и чтобы его преодолеть, вы должны суметь где-то создать текст программы, успешно его скомпилировать, загрузить, запустить на выполнение и разобраться, куда будет отправлен результат. Как только вы овладеете этим, все остальное окажется относительно просто.

Си-программа, печатающая "здравствуй, мир", выглядит так:

```
#include <stdio.h>
main()
{
  printf("здравствуй, мир\n");
}
```

Как запустить эту программу, зависит от системы, которую вы используете. Так, в операционной системе UNIX необходимо сформировать исходную программу в файле с именем, заканчивающимся символами ". с", например в файле hello. с, который затем компилируется с помощью команды

CGhello.c

Если вы все сделали правильно - не пропустили где-либо знака и не допустили орфографических ошибок, то компиляция пройдет "молча" и вы получите файл, готовый к исполнению и названный a.out. Если вы теперь запустите этот файл на выполнение командой

a.out

программа напечатает

здравствуй, мир

В других системах правила запуска программы на выполнение могут быть иными; чтобы узнать о них, поговорите со специалистами.

Теперь поясним некоторые моменты, касающиеся самой программы. Программа на Си, каких бы размеров она ни была, состоит из *функций* и *переменных.* Функции содержат *инструкции,* описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений. Функции в Си похожи на под-

программы и функции Фортрана или на процедуры и функции Паскаля. Приведенная программа - это функция *с* именем main. Обычно вы вольны придумывать любые имена для своих функций, но "main" - особое имя: любая программа начинает свои вычисления с первой инструкции функции main.

Обычно main для выполнения своей работы пользуется услугами других функций; одни из них пишутся самим программистом, а другие берутся готовыми из имеющихся в его распоряжении библиотек. Первая строка программы:

```
#include <stdio.h>
```

сообщает компилятору, что он должен включить информацию о стандартной библиотеке ввода-вывода. Эта строка встречается в начале многих исходных файлов Си-программ. Стандартная библиотека описана в главе 7 и приложении В.

Один из способов передачи данных между функциями состоит в том, что функция при обращении к другой функции передает ей список значений, называемых *аргументами*. Этот список берется в скобки и помещается после имени функции. В нашем примере main определена как функция, которая не ждет никаких аргументов, что отмечено пустым списком ().

Первая программа на Си

`#include <stdio.h>`	*Включение информации о стандартной библиотеке.*
`main()`	*Определение функции с именем* main, *не получающей никаких аргументов.*
`{`	*Инструкции* main *заключаются в фигурные скобки.*
`printf ("здравствуй, мир\n");`	*Функция* main *вызывает библиотечную функцию* printf *для печати заданной последовательности символов;* \n - *символ новой строки.*
`}`	

Инструкции функции заключаются в фигурные скобки {}. Функция main содержит только одну инструкцию

```
printf ("здравствуй, мир\n");
```

Функция вызывается по имени, после которого, в скобках, указывается список аргументов. Таким образом, приведенная выше строка - это вызов

функции printf с аргументом "здравствуй, мир\n". Функция printf – это библиотечная функция, которая в данном случае напечатает последовательность символов, заключенную в двойные кавычки.

Последовательность символов в двойных кавычках, такая как "здравствуй, мир\n", называется *строкой символов,* или *строковой константой.* Пока что в качестве аргументов для printf и других функций мы будем использовать только строки символов.

В Си комбинация \n внутри строки символов обозначает *символ новой строки* и при печати вызывает переход к левому краю следующей строки. Если вы удалите \n (стоит поэкспериментировать), то обнаружите, что, закончив печать, машина не переходит на новую строку. Символ новой строки в текстовый аргумент printf следует включать явным образом. Если вы попробуете выполнить, например,

```
printf("здравствуй, мир
");
```

компилятор выдаст сообщение об ошибке.

Символ новой строки никогда не вставляется автоматически, так что одну строку можно напечатать по шагам с помощью нескольких обращений к printf. Нашу первую программу можно написать и так:

```
flinclude <stdio.h>

main()
{
    printf("здравствуй, ");
    printf("мир");
    printf('\n');
}
```

В результате ее выполнения будет напечатана та же строка, что и раньше.

Заметим, что \n обозначает только один символ. Такие особые комбинации символов, начинающиеся с обратной наклонной черты, как \n, и называемые *эскейп-последовательностями,* широко применяются для обозначения трудно представимых или невидимых символов. Среди прочих в Си имеются символы \t, \b, \", \\, обозначающие соответственно табуляцию, возврат на один символ назад ("забой" последнего символа), двойную кавычку, саму наклонную черту. Полный список таких символов представлен в параграфе 2.3.

Упражнение 1.1. Выполните программу, печатающую "*здравствуй, мир*", в вашей системе. Поэкспериментируйте, удаляя некоторые части программы, и посмотрите, какие сообщения об ошибках вы получите.

Упражнение 1.2. Выясните, что произойдет, если в строковую константу аргумента `printf` вставить `\c`, где с - символ, не входящий в представленный выше список.

1.2. Переменные и арифметические выражения

Приведенная ниже программа выполняет вычисления по формуле $^\circ C = (5/9)\ (T-32)$ и печатает таблицу соответствия температур по Фаренгейту температурам по Цельсию:

0	-17
20	-6
40	4
60	15
80	26
100	37
120	48
140	60
160	71
180	82
200	93
220	104
240	115
260	126
280	137
300	148

Как и предыдущая, эта программа состоит из определения одной-единственной функции main. Она длиннее программы, печатающей "здравствуй, мир", но по сути не сложнее. На ней мы продемонстрируем несколько новых возможностей, включая комментарий, объявления, переменные, арифметические выражения, циклы и форматный вывод.

```
#include <stdio.h>

/* печать таблицы температур по Фаренгейту
и Цельсию для fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* нижняя граница таблицы температур */
    upper = 300;    /* верхняя граница */
```

```
    step = 20;      /* шаг */

    fahr = lower;
    while (fahr <= upper) {
        Celsius = 5 * (fahr-32) / 9;
        printf ("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Две строки:

```
/* печать таблицы температур по Фаренгейту
   и Цельсию для fahr = 0, 20 . . . . 300 */
```

являются *комментарием,* который в данном случае кратко объясняет, что делает программа. Все символы, помещенные между /* и */, игнорируются компилятором, и этим можно свободно пользоваться, чтобы сделать программу более понятной. Комментарий можно располагать в любом месте, где могут стоять символы пробела, табуляции или символ новой строки.

В Си любая переменная должна быть объявлена раньше, чем она будет использована; обычно все переменные объявляются в начале функции перед первой исполняемой инструкцией. В *объявлении* описываются свойства переменных. Оно состоит из названия типа и списка переменных, например:

```
int fahr, celsius;
int lower, upper, step;
```

Тип int означает, что значения перечисленных переменных есть целые, в отличие от него тип float указывает на значения с плавающей точкой, т. е. на числа, которые могут иметь дробную часть. Диапазоны значений обоих типов зависят от используемой машины.

Числа типа int бывают как 16-разрядные (лежат в диапазоне от -32768 до 32767), так и 32-разрядные. Числа типа float обычно представляются 32-разрядными словами, имеющими по крайней мере 6 десятичных значащих цифр (лежат приблизительно в диапазоне от 10^{-38} до 10^{+38}).

Помимо int и float в Си имеется еще несколько базовых типов для данных, это:

char	- символ - единичный байт;
short	- короткое целое;
long	- длинное целое;
double	- с плавающей точкой с двойной точностью.

Размеры объектов указанных типов также зависят от машины. Из базовых типов можно создавать: *массивы, структуры* и *объединения, указатели* на объекты базовых типов и функции, возвращающие значения этих типов в качестве результата. Обо всем этом мы расскажем позже.

Вычисления в программе преобразования температур начинаются с *инструкций присваивания:*

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

которые устанавливают указанные в них переменные в начальные значения. Любая инструкция заканчивается точкой с запятой.

Все строки таблицы вычисляются одним и тем же способом, поэтому мы воспользуемся циклом, повторяющим это вычисление для каждой строки. Необходимые действия выполнит цикл while:

```
while (fahr <= upper) {
    ...
}
```

Он работает следующим образом. Проверяется условие в скобках. Если оно истинно (значение fahr меньше или равно значению upper), то выполняется тело цикла (три инструкции, заключенные в фигурные скобки). Затем опять проверяется условие, и если оно истинно, то тело цикла выполняется снова. Когда условие становится ложным (fahr превысило upper), цикл завершается, и вычисления продолжаются с инструкции, следующей за циклом. Поскольку никаких инструкций за циклом нет, программа завершает работу.

Телом цикла while может быть одна или несколько инструкций, заключенные в фигурные скобки, как в программе преобразования температур, или одна-единственная инструкция без скобок, как в цикле

```
while (i < j)
    i = 2 •* i;
```

И в том и в другом случае инструкции, находящиеся под управлением while, мы будем записывать со сдвигом, равным одной позиции табуляции, которая в программе указывается четырьмя пробелами; благодаря этому будут ясно видны инструкции, расположенные внутри цикла. Отступы подчеркивают логическую структуру программы. Си-компилятор не обращает внимания на внешнее оформление программы, но наличие в нужных местах отступов и пробелов существенно влияет на то, насколько легко она будет восприниматься человеком при просмотре. Чтобы лучше

была видна логическая структура выражения, мы рекомендуем на каждой строке писать только по одной инструкции и с обеих сторон от операторов ставить пробелы. Положение скобок не так важно, хотя существуют различные точки зрения на этот счет. Мы остановились на одном из нескольких распространенных стилей их применения. Выберите тот, который больше всего вам нравится, и строго ему следуйте.

Бóльшая часть вычислений выполняется в теле цикла. Температура по Фаренгейту переводится в температуру по Цельсию и присваивается переменной celsius посредством инструкции

```
Celsius = 5 * (fahr-32) / 9;
```

Причина, по которой мы сначала умножаем на 5 и затем делим на 9, а не сразу умножаем на 5/9, связана с тем, что в Си, как и во многих других языках, деление целых сопровождается *отбрасыванием,* т. е. потерей дробной части. Так как 5 и 9 - целые, отбрасывание в 5/9 дало бы нуль, и на месте температур по Цельсию были бы напечатаны нули.

Этот пример прибавил нам еще немного знаний о том, как работает функция printf. Функция printf - это универсальная функция форматного ввода-вывода, которая будет подробно описана в главе 7. Ее первый аргумент - строка символов, в которой каждый символ % соответствует одному из последующих аргументов (второму, третьему, ...), а информация, расположенная за символом %, указывает на вид, в котором выводится каждый из этих аргументов. Например, %d специфицирует выдачу аргумента в виде целого десятичного числа, и инструкция

```
printf("%d\t%d\n", fahr, celsius);
```

печатает целое fahr, выполняет табуляцию (\t) и печатает целое celsius.

В функции printf каждому спецификатору первого аргумента (конструкции, начинающейся с %) соответствует второй аргумент, третий аргумент и т. д. Спецификаторы и соответствующие им аргументы должны быть согласованы по количеству и типам: в противном случае напечатано будет не то, что нужно.

Кстати, printf не является частью языка Си, и вообще в языке нет никаких специальных конструкций, определяющих ввод-вывод. Функция printf -- лишь полезная функция стандартной библиотеки, которая обычно доступна для Си-программ. Поведение функции printf, однако, оговорено стандартом ANSI, и ее свойства должны быть одинаковыми во всех Си-системах, удовлетворяющих требованиям стандарта.

Желая сконцентрировать ваше внимание на самом Си, мы не будем много говорить о вводе-выводе до главы 7. В частности, мы отложим разговор о форматном вводе. Если вам потребуется ввести числа, советуем про-

читать в параграфе 7.4 то, что касается функции scanf. Эта функция отличается от printf лишь тем, что она вводит данные, а не выводит.

Существуют еще две проблемы, связанные с программой преобразования температур. Одна из них (более простая) состоит в том, что выводимый результат выглядит несколько неряшливо, поскольку числа не выровнены по правой позиции колонок. Это легко исправить, добавив в каждый из спецификаторов формата %d указание о ширине поля; при этом программа будет печатать числа, прижимая их к правому краю указанных полей. Например, мы можем написать

```
printf("%3d %6d\n", fahr, Celsius);
```

чтобы в каждой строке первое число печатать в поле из трех позиций, а второе - из шести. В результате будет напечатано:

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
...
```

Вторая, более серьезная проблема связана с тем, что мы пользуемся целочисленной арифметикой и поэтому не совсем точно вычисляем температуры по шкале Цельсия. Например, 0 *°F* на самом деле (с точностью до десятой) равно -17.8 *°C*, а не -17. Чтобы получить более точные значения температур, нам надо пользоваться не целочисленной арифметикой, а арифметикой с плавающей точкой. Это потребует некоторых изменений в программе.

```
#include <stdio.h>

/* печать таблицы температур по Фаренгейту и Цельсию для
   fahr = 0, 20 . . . 300; вариант с плавающей точкой */
main()
{
   float fahr, Celsius;
   int lower, upper; step;

   lower = 0;      /* нижняя граница таблицы температур */
   upper = 300;    /* верхняя граница */
   step = 20;      /* шаг */

   fahr = lower;
   while (fahr <= upper) {
```

```
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Программа мало изменилась. Она отличается от предыдущей лишь тем, что fahr и celsius объявлены как float, а формула преобразования написана в более естественном виде. В предыдущем варианте нельзя было писать 5/9, так как целочисленное деление в результате обрезания дало бы нуль. Десятичная точка в константе указывает на то, что последняя рассматривается как число с плавающей точкой, и 5.0/9.0, таким образом, есть частное от деления двух значений с плавающей точкой, которое не предполагает отбрасывания дробной части. В том случае, когда арифметическая операция имеет целые операнды, она выполняется по правилам целочисленной арифметики. Если же один операнд с плавающей точкой, а другой - целый, то перед тем, как операция будет выполнена, последний будет преобразован в число с плавающей точкой. Если бы мы написали fahr-32, то 32 автоматически было бы преобразовано в число с плавающей точкой. Тем не менее при записи констант с плавающей точкой мы всегда используем десятичную точку, причем даже в тех случаях, когда константы на самом деле имеют целые значения. Это делается для того, чтобы обратить внимание читающего программу на их природу.

Более подробно правила, определяющие, в каких случаях целые переводятся в числа с плавающей точкой, рассматриваются в главе 2. А сейчас заметим, что присваивание

```
fahr = lower;
```

и проверка

```
while (fahr <= upper)
```

работают естественным образом, т. е. перед выполнением операции значение int приводится к float.

Спецификация %3.0f в printf определяет печать числа с плавающей точкой (в данном случае числа fahr) в поле шириной не более трех позиций без десятичной точки и дробной части. Спецификация %6.1f описывает печать другого числа (celsius) в поле из шести позиций с одной цифрой после десятичной точки. Напечатано будет следующее:

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

Ширину и точность можно не задавать: %6f означает, что число будет занимать не более шести позиций; %.2f - число имеет две цифры после десятичной точки, но ширина не ограничена; %f просто указывает на печать числа с плавающей точкой.

%d	- печать десятичного целого.
%6d	- печать десятичного целого в поле из шести позиций.
%f	- печать числа с плавающей точкой.
%6f	- печать числа с плавающей точкой в поле из шести позиций.
%.2f	- печать числа с плавающей точкой с двумя цифрами после десятичной точки.
%6.2f	- печать числа с плавающей точкой и двумя цифрами после десятичной точки в поле из шести позиций.

Кроме того, printf допускает следующие спецификаторы: %o для восьмеричного числа; %x для шестнадцатеричного числа; %c для символа; %s для строки символов и %% для самого %.

Упражнение 1.3. Усовершенствуйте программу преобразования температур таким образом, чтобы над таблицей она печатала заголовок.

Упражнение 1.4. Напишите программу, которая будет печатать таблицу соответствия температур по Цельсию температурам по Фаренгейту.

1.3. Инструкция for

Существует много разных способов для написания одной и той же программы. Видоизменим нашу программу преобразования температур:

```
#include <stdio.h>

/* печать таблицы температур по Фаренгейту и Цельсию */
main()
{
  int fahr;

  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
  printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Эта программа печатает тот же результат, но выглядит она, несомненно, по-другому. Главное отличие заключается в отсутствии большинства переменных. Осталась только переменная fahr, которую мы объявили как int. Нижняя и верхняя границы и шаг присутствуют в виде констант

в инструкции for - новой для нас конструкции, а выражение, вычисляющее температуру по Цельсию, теперь задано третьим аргументом функции printf, а не в отдельной инструкции присваивания.

Последнее изменение является примером применения общего правила: в любом контексте, где возможно использовать значение переменной какого-то типа, можно использовать более сложное выражение того же типа. Так, на месте третьего аргумента функции printf согласно спецификатору %6.1f должно быть значение с плавающей точкой, следовательно, здесь может быть любое выражение этого типа.

Инструкция for описывает цикл, который является обобщением цикла while. Если вы сравните его с ранее написанным while, то вам станет ясно, как он работает. Внутри скобок имеются три выражения, разделяемые точкой с запятой. Первое выражение - инициализация

```
fahr = 0
```

выполняется один раз перед тем, как войти в цикл. Второе - проверка условия продолжения цикла

```
fahr <= 300
```

Условие вычисляется, и если оно истинно, выполняется тело цикла (в нашем случае это одно обращение к printf). Затем осуществляется приращение шага:

```
fahr = fahr + 20
```

и условие вычисляется снова. Цикл заканчивается, когда условие становится ложным. Как и в случае с while, тело for-цикла может состоять из одной инструкции или из нескольких, заключенных в фигурные скобки. На месте этих трех выражений (инициализации, условия и приращения шага) могут стоять произвольные выражения.

Выбор между while и for определяется соображениями ясности программы. Цикл for более удобен в тех случаях, когда инициализация и приращение шага логически связаны друг с другом общей переменной и выражаются единичными инструкциями, поскольку названный цикл компактнее цикла while, а его управляющие части сосредоточены в одном месте.

Упражнение 1.5. Измените программу преобразования температур так, чтобы она печатала таблицу в обратном порядке, т. е. от 300 до 0.

1.4. Именованные константы

Прежде чем мы закончим рассмотрение программы преобразования температур, выскажем еще одно соображение. Очень плохо, когда по про-

грамме рассеяны "загадочные числа", такие как 300, 20. Тот, кто будет читать программу, не найдет в них и намека на то, что они собой представляют. Кроме того, их трудно заменить на другие каким-то систематическим способом. Одна из возможностей справиться с такими числами - дать им осмысленные имена. Строка #define определяет *символьное имя*, или *именованную константу,* для заданной строки символов:

#define *имя подставляемый-текст*

С этого момента при любом появлении *имени* (если только оно встречается не в тексте, заключенном в кавычки, и не является частью определения другого имени) оно будет заменяться на соответствующий ему *подставляемый-текст. Имя* имеет тот же вид, что и переменная: последовательность букв и цифр, начинающаяся с буквы. *Подставляемый-текст* может быть любой последовательностью символов, среди которых могут встречаться не только цифры.

```
#include <stdio.h>

#define LOWER 0     /* нижняя граница таблицы */
#define UPPER 300   /* верхняя граница */
#define STEP 20     /* размер шага */

/* печать таблицы температур по Фаренгейту и Цельсию */
main()
{
  int fahr;

  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
    printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Величины LOWER, UPPER и STEP - именованные константы, а не переменные, поэтому для них нет объявлений. По общепринятому соглашению имена именованных констг.нт набираются заглавными буквами, чтобы они отличались от обычных переменных, набираемых строчными. Заметим, что в конце «define-строки точка с запятой не ставится.

1.5. Ввод-вывод символов

Теперь мы намерены рассмотреть семейство программ по обработке текстов. Вы обнаружите, что многие существующие программы являются просто расширенными версиями обсуждаемых здесь прототипов.

Стандартная библиотека поддерживает очень простую модель ввода-вывода. Текстовый ввод-вывод вне зависимости от того, откуда он исходит или куда направляется, имеет дело с потоком символов. *Текстовый поток* - это последовательность символов, разбитая на строки, каждая из которых содержит нуль или более символов и завершается символом новой строки. Обязанность следить за тем, чтобы любой поток ввода-вывода отвечал этой модели, возложена на библиотеку: программист, пользуясь библиотекой, не должен заботиться о том, в каком виде строки представляются вне программы.

Стандартная библиотека включает несколько функций для чтения и записи одного символа. Простейшие из них - getchar и putchar. За одно обращение к getchar считывается *следующий символ ввода* из текстового потока, и этот символ выдается в качестве результата. Так, после выполнения

```
c = getchar()
```

переменная с содержит очередной символ ввода. Обычно символы поступают с клавиатуры. Ввод из файлов рассматривается в главе 7.

Обращение к putchar приводит к печати одного символа. Так,

```
putchar(c)
```

напечатает содержимое целой переменной с в виде символа (обычно на экране). Вызовы putchar и printf могут произвольным образом перемежаться. Вывод будет формироваться в том же порядке, что и обращения к этим функциям.

1.5.1. Копирование файла

При наличии функций getchar и putchar, ничего больше не зная о вводе-выводе, можно написать удивительно много полезных программ. Простейший пример - это программа, копирующая по одному символу с входного потока в выходной поток:

```
чтение символа
while (символ не является признаком конца файла)
    вывод только что прочитанного символа
    чтение символа
```

Оформляя ее в виде программы на Си, получим

```
#include <stdio.h>

/* копирование ввода на вывод; 1-я версия */
main()
{
    int c;
```

```
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Оператор отношения != означает "не равно".

Каждый символ, вводимый с клавиатуры или появляющийся на экране, как и любой другой символ внутри машины, кодируется комбинацией битов. Тип char специально предназначен для хранения символьных данных, однако для этого также годится и любой целый тип. Мы пользуемся типом int и делаем это по одной важной причине, которая требует разъяснений.

Существует проблема: как отличить конец ввода от обычных читаемых данных. Решение заключается в том, чтобы функция getchar по исчерпании входного потока выдавала в качестве результата такое значение, которое нельзя было бы спутать ни с одним реальным символом. Это значение есть EOF (аббревиатура от *end of file* - конец файла). Мы должны объявить переменную с такого типа, чтобы его "хватило" для представления всех возможных результатов, выдаваемых функцией getchar. Нам не подходит тип char, так как с должна быть достаточно "емкой", чтобы помимо любого значения типа char быть в состоянии хранить и EOF. Вот почему мы используем int, а не char.

EOF - целая константа, определенная в <stdio.h>. Какое значение имеет эта константа - неважно, лишь бы оно отличалось от любого из возможных значений типа char. Использование именованной константы с унифицированным именем гарантирует, что программа не будет зависеть от конкретного числового значения, которое, возможно, в других Си-системах будет иным.

Программу копирования можно написать более сжато. В Си любое присваивание, например

```
c = getchar()
```

трактуется как выражение со значением, равным значению левой части после присваивания. Это значит, что присваивание может встречаться внутри более сложного выражения. Если присваивание переменной с расположить в проверке условия цикла while, то программу копирования можно будет записать в следующем виде:

```
#include <stdio.h>

/* копирование ввода на вывод; 2-я версия */
```

```
main()
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(c);
}
```

Цикл while, пересылая в с полученное от getchar значение, сразу же проверяет: не является ли оно "концом файла". Если это не так, выполняется тело цикла while и печатается символ. По окончании ввода завершается работа цикла while, а тем самым и main.

В данной версии ввод "централизован" - в программе имеется только одно обращение к getchar. В результате она более компактна и легче воспринимается при чтении. Вам часто придется сталкиваться с такой формой записи, где присваивание делается вместе с проверкой. (Чрезмерное увлечение ею, однако, может запутать программу, поэтому мы постараемся пользоваться указанной формой разумно.)

Скобки внутри условия, вокруг присваивания, необходимы. *Приоритет* != выше, чем приоритет =, из чего следует, что при отсутствии скобок проверка != будет выполняться до операции присваивания =. Таким образом, запись

```
c = getchar() != EOF
```

эквивалентна записи

```
c = (getchar() != EOF)
```

А это совсем не то, что нам нужно: переменной с будет присваиваться 0 или 1 в зависимости от того, встретит или не встретит getchar признак конца файла. (Более подробно об этом см. в главе 2.)

Упражнение **1.6.** Убедитесь в том, что выражение getchar() != EOF получает значение 0 или 1.

Упражнение 1.7. Напишите программу, печатающую значение EOF.

1.5.2. Подсчет символов

Следующая программа занимается подсчетом символов; она имеет много сходных черт с программой копирования.

```
#include <stdio.h>

/* подсчет вводимых символов; 1-я версия */
main()
```

```
{
    long nc;
    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Инструкция

```
++nc;
```

представляет новый оператор ++, который означает *увеличить на единицу*. Вместо этого можно было бы написать nc = nc+1, но ++nc намного короче, а часто и эффективнее. Существует аналогичный оператор --, означающий *уменьшить на единицу*. Операторы ++ и -- могут быть как префиксными (++nc), так и постфиксными (nc++). Как будет показано в главе 2, эти две формы в выражениях имеют разные значения, но и ++nc, и nc++ добавляют к nc единицу. В данном случае мы остановились на префиксной записи.

Программа подсчета символов накапливает сумму в переменной типа long. Целые типа long имеют не менее 32 битов. Хотя на некоторых машинах типы int и long имеют одинаковый размер, существуют, однако, машины, в которых int занимает 16 битов с максимально возможным значением 32767, а это - сравнительно маленькое число, и счетчик типа int может переполниться. Спецификация %ld в printf указывает, что сортветствующий аргумент имеет тип long.

Возможно охватить еще больший диапазон значений, если использовать тип double (т. е. float с двойной точностью). Применим также инструкцию for вместо while, чтобы продемонстрировать другой способ написания цикла.

```
#include <stdio.h>

/* подсчет вводимых символов; 2-я версия */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

В printf спецификатор %f применяется как для float, так и для double; спецификатор %.0f означает печать без десятичной точки и дробной части (последняя в нашем случае отсутствует).

Тело указанного for-цикла пусто, поскольку кроме проверок и приращений счетчика делать ничего не нужно. Но правила грамматики Си требуют, чтобы for-цикл имел тело. Выполнение этого требования обеспечивает изолированная точка с запятой, называемая *пустой инструкцией.* Мы поставили точку с запятой на отдельной строке для большей наглядности.

Наконец, заметим, что если ввод не содержит ни одного символа, то при первом же обращении к getchar условие в while или for не будет выполнено и программа выдаст нуль, что и будет правильным результатом. Это важно. Одно из привлекательных свойств циклов while и for состоит в том, что условие проверяется до того, как выполняется тело цикла. Если ничего делать не надо, то ничего делаться и не будет, пусть даже тело цикла не выполнится ни разу. Программа должна вести себя корректно и при нулевом количестве вводимых символов. Само устройство циклов while и for дает дополнительную уверенность в правильном поведении программы в случае граничных условий.

1.5.3. Подсчет строк

Следующая программа подсчитывает строки. Как упоминалось выше, стандартная библиотека обеспечивает такую модель ввода-вывода, при которой входной текстовый поток состоит из последовательности строк, каждая из которых заканчивается символом новой строки. Следовательно, подсчет строк сводится к подсчету числа символов новой строки.

```
#include <stdio.h>

/* подсчет строк входного потока */
main()
{
  int c, nl;

  nl = 0;
  while ((c = getchar()) != EOF)
    if (c == V)
      ++nl;
  printf("%d\n", nl);
}
```

Тело цикла теперь образует инструкция if, под контролем которой находится увеличение счетчика nl на единицу. Инструкция if проверяет условие в скобках и, если оно истинно, выполняет следующую за ним инструкцию (или группу инструкций, заключенную в фигурные скобки). Мы опять делаем отступы в тексте программы, чтобы показать, что чем управляется.

Двойной знак равенства в языке Си обозначает оператор "равно" (он аналогичен оператору = в Паскале и . EQ. в Фортране). Удваивание знака = в операторе проверки на равенство сделано для того, чтобы отличить его от единичного =, используемого в Си для обозначения присваивания. Предупреждаем: начинающие программировать на Си иногда пишут =, а имеют в виду ==. Как мы увидим в главе 2, в этом случае результатом будет обычно вполне допустимое по форме выражение, на которое компилятор не выдаст никаких предупреждающих сообщений[1].

Символ, заключенный в одиночные кавычки, представляет собой целое значение, равное коду этого символа (в кодировке, принятой на данной машине). Это так называемая *символьная константа*. Существует и другой способ для написания маленьких целых значений. Например, 'A' есть символьная константа; в наборе символов ASCII ее значение равняется 65 - внутреннему представлению символа A. Конечно, 'A' в роли константы предпочтительнее, чем 65, поскольку смысл первой записи более очевиден, и она не зависит от конкретного способа кодировки символов.

Эскейп-последовательности, используемые в строковых константах, допускаются также и в символьных константах. Так, '\n' обозначает код символа новой строки, который в ASCII равен 10. Следует обратить особое внимание на то, что '\n' обозначает один символ (код которого в выражении рассматривается как целое значение), в то время как "\n" - строковая константа, в которой чисто случайно указан один символ. Более подробно различие между символьными и строковыми константами разбирается в главе 2.

Упражнение 1.8. Напишите программу для подсчета пробелов, табуляций и новых строк.

Упражнение 1.9. Напишите программу, копирующую символы ввода в выходной поток и заменяющую стоящие подряд пробелы на один пробел.

Упражнение 1.10. Напишите программу, копирующую вводимые символы в выходной поток с заменой символа табуляции на \t, символа забоя на \b и каждой обратной наклонной черты на \\. Это сделает видимыми все символы табуляции и забоя.

1.5.4. Подсчет слов

Четвертая из нашей серии полезных программ подсчитывает строки, слова и символы, причем под словом здесь имеется в виду любая строка

[1] Современные компиляторы, как правило, выдают предупреждение о возможной ошибке. - *Примеч. ред.*

символов, не содержащая в себе пробелов, табуляций и символов новой строки. Эта программа является упрощенной версией программы wc системы UNIX.

```
#include <stdio.h>

#define IN 1        /* внутри слова */
#define OUT 0       /* вне слова */

/* подсчет строк, слов и символов */
main()
{
    int c, nl, nw, nc, state;

    state=OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == 'n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t' )
            state=OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Каждый раз, встречая первый символ слова, программа изменяет значение счетчика слов на 1. Переменная state фиксирует текущее состояние - находимся мы внутри или вне слова. Вначале ей присваивается значение OUT, что соответствует состоянию "вне слова". Мы предпочитаем пользоваться именованными константами IN и OUT, а не собственно значениями 1 и 0, чтобы сделать программу более понятной. В такой маленькой программе этот прием мало что дает, но в большой программе увеличение ее ясности окупает незначительные дополнительные усилия, потраченные на то, чтобы писать программу в таком стиле с самого начала. Вы обнаружите, что большие изменения гораздо легче вносить в те программы, в которых магические числа встречаются только в виде именованных констант.

Строка

```
nl = nw = nc = 0;
```

устанавливает все три переменные в нуль. Такая запись не является какой-то особой конструкцией и допустима потому, что присваивание есть выражение со своим собственным значением, а операции присваивания выполняются справа налево. Указанная строка эквивалентна

```
nl = (nw = (nc = 0));
```

Оператор !! означает ИЛИ, так что строка

```
if (c == ' ' !! c == '\n' !! c == '\t')
```

читается как "если c есть пробел, *или* c есть новая строка, *или* c есть табуляция". (Напомним, что видимая эскейп-последовательность \t обозначает символ табуляции.) Существует также оператор &&, означающий И. Его приоритет выше, чем приоритет !!. Выражения, связанные операторами && или !!, вычисляются слева направо; при этом гарантируется, что вычисления сразу прервутся, как только будет установлена истинность или ложность условия. Если c есть пробел, то дальше проверять, является значение c символом новой строки или же табуляции, не нужно. В этом частном случае данный способ вычислений не столь важен, но он имеет значение в более сложных ситуациях, которые мы вскоре рассмотрим.

В примере также встречается слово else, которое указывает на альтернативные действия, выполняемые в случае, когда условие, указанное в if, не является истинным. В общем виде условная инструкция записывается так:

if *(выражение)*
   инструкция$_1$
else
   инструкция$_2$

В конструкции if-else выполняется одна и только одна из двух инструкций. Если *выражение* истинно, то выполняется *инструкция*$_1$, если нет, то - *инструкция*$_2$. Каждая из этих двух *инструкций* представляет собой либо одну инструкцию, либо несколько, заключенных в фигурные скобки. В нашей программе после else стоит инструкция if, управляющая двумя такими инструкциями.

Упражнение 1.11. Как протестировать программу подсчета слов? Какой ввод вероятнее всего обнаружит ошибки, если они были допущены?

Упражнение 1.12. Напишите программу, которая печатает содержимое своего ввода, помещая по одному слову на каждой строке.

1.6. Массивы

А теперь напишем программу, подсчитывающую по отдельности каждую цифру, символы-разделители (пробелы, табуляции и новые-строки) и все другие символы. Это несколько искусственная программа, но она позволит нам в одном примере продемонстрировать еще несколько возможностей языка Си. Имеется двенадцать категорий вводимых символов. Удобно все десять счетчиков цифр хранить в массиве, а не в виде десяти отдельных переменных. Вот один из вариантов этой программы:

```
ttinclude <stdio.h>

/* подсчет цифр, символов-разделителей и прочих символов */
main()
{
  int c, i, nwhite, nother;
  int ndigit[10];

  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit[i]= 0;

  while ((c = getchar()) != EOF)
    if (c >= '0' && c <= '9' )
      ++ndigit[c - '0'];
    else if (c == ' ' !! c == '\n' !! c == '\t')
      ++nwhite;
    else
      ++nother;

  printf ("цифры =");
  for (i = 0; i < 10; ++i)
    printf(" %d", ndigit[i]);
  printf(", символы-разделители = %d, прочие = %d\n", nwhite, nother);
}
```

В результате выполнения этой программы будет напечатан следующий результат:

```
цифры = 9 3 0 0 0 0 0 0 0 1 , символы-разделители = 123, прочие = 345
```

Объявление

```
int ndigit[10];
```

объявляет ndigit массивом из 10 значений типа int. В Си элементы массива всегда нумеруются начиная с нуля, так что элементами этого массива будут ndigit[0], ndigit[1], ..., ndigit[9], что учитывается в for-циклах (при инициализации и печати массива).

Индексом может быть любое целое выражение, образуемое целыми переменными (например i) и целыми константами.

Приведенная программа опирается на определенные свойства кодировки цифр. Например, проверка

```
if (c >= '0' && c <= '9') ...
```

определяет, является ли находящийся в c символ цифрой. Если это так, то

```
c - '0'
```

есть числовое значение цифры. Сказанное справедливо только в том случае, если для ряда значений '0', '1', ..., '9' каждое следующее значение на 1 больше предыдущего. К счастью, это правило соблюдается во всех наборах символов.

По определению, значения типа char являются просто малыми целыми, так что переменные и константы типа char в арифметических выражениях идентичны значениям типа int. Это и естественно, и удобно; например, c-'0' есть целое выражение с возможными значениями от 0 до 9, которые соответствуют символам от '0' до '9', хранящимся в переменной c. Таким образом, значение данного выражения является правильным индексом для массива ndigit.

Следующий фрагмент определяет, является символ цифрой, символом-разделителем или чем-нибудь иным.

```
if (c >= '0' && c <= '9')
  ++ndigit[c - '0'];
else if (c ==' ' || c == '\n' || c == '\t')
  ++nwhite;
else
  ++nother
```

Конструкция вида

```
if  (условие₁)
  инструкция₁
else  if  (условие₂)
  инструкция₂
...
else
  инструкцияₙ
```

часто применяется для выбора одного из нескольких альтернативных путей, имеющихся в программе. *Условия* вычисляются по порядку в направлении сверху вниз до тех пор, пока одно из них не будет удовлетворено; в этом случае будет выполнена соответствующая ему *инструкция,* и работа всей конструкции завершится. (Любая из *инструкций* может быть группой инструкций в фигурных скобках.) Если ни одно из *условий* не удовлетворено, выполняется последняя *инструкция,* расположенная сразу за else, если таковая имеется. Если же else и следующей за ней *инструкции* нет (как это было в программе подсчета слов), то никакие действия вообще не производятся. Между первым if и завершающим else может быть сколько угодно комбинаций вида

```
else if (условие)
    инструкция
```

Когда их несколько, программу разумно форматировать так, как мы здесь показали. Если же каждый следующий if сдвигать вправо относительно предыдущего else, то при длинном каскаде проверок текст окажется слишком близко прижатым к правому краю страницы.

Инструкция switch, речь о которой пойдет в главе 3, обеспечивает другой способ изображения многопутевого ветвления на языке Си. Он более подходит, в частности, тогда, когда условием перехода служит совпадение значения некоторого выражения целочисленного типа с одной из констант, входящих в заданный набор. Вариант нашей программы, реализованной с помощью switch, приводится в параграфе 3.4.

Упражнение 1.13. Напишите программу, печатающую гистограммы длин вводимых слов. Гистограмму легко рисовать горизонтальными полосами. Рисование вертикальными полосами - более трудная задача.

Упражнение 1.14. Напишите программу, печатающую гистограммы частот встречаемости вводимых символов.

1.7. Функции

Функции в Си играют ту же роль, что и подпрограммы и функции в Фортране или процедуры и функции в Паскале. Функция обеспечивает удобный способ отдельно оформить некоторое вычисление и пользоваться им далее, не заботясь о том, как оно реализовано. После того, как функции написаны, можно забыть, *как* они сделаны, достаточно знать лишь, *что* они умеют делать. Механизм использования функции в Си удобен, легок и эффективен. Нередко вы будете встречать короткие функции, вы-

зываемые лишь единожды; они оформлены в виде функции с одной-единственной целью - получить более ясную программу.

До сих пор мы пользовались готовыми функциями вроде main, getchar и putchar, теперь настала пора нам самим написать несколько функций. В Си нет оператора возведения в степень вроде ** в Фортране. Поэтому проиллюстрируем механизм определения функции на примере функции power(m, n), которая возводит целое m в целую положительную степень n. Так, power(2, 5) имеет значение 32. На самом деле для практического применения эта функция малопригодна, так как оперирует лишь малыми целыми степенями, однако она вполне может послужить иллюстрацией. (В стандартной библиотеке есть функция pow(x, y), вычисляющая x^y.)

Итак, мы имеем функцию power и главную функцию main, пользующуюся ее услугами, так что вся программа выглядит следующим образом:

```
#include <stdio.h>

int power(int m, int n);

/* тест функции power */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3, i));
    return 0;
}

/* возводит base в n-ю степень; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Определение любой функции имеет следующий вид:

тип-результата имя-функции (список параметров, если он есть)
{
  объявления
  инструкции
}

Определения функций могут располагаться в любом порядке в одном или в нескольких исходных файлах, но любая функция должна быть целиком расположена в каком-то одном. Если исходный текст программы распределен по нескольким файлам, то, чтобы ее скомпилировать и загрузить, вам придется сказать несколько больше, чем при использовании одного файла; но это уже относится к операционной системе, а не к языку. Пока мы предполагаем, что обе функции находятся в одном файле, так что будет достаточно тех знаний, которые вы уже получили относительно запуска программ на Си.

В следующей строке из функции main к power обращаются дважды.

```
printf("%d %d %d\n", i, power(2,i), power(-3,1));
```

При каждом вызове функции power передаются два аргумента, и каждый раз главная профамма main в ответ получает целое число, которое затем приводится к должному формату и печатается. Внутри выражения power(2,1) представляет собой целое значение точно так же, как 2 или i. (Не все функции в качестве результата выдают целые значения; подробно об этом будет сказано в главе 4.)

В первой строке определения power:

```
int power(int base, int n)
```

указываются типы параметров, имя функции и тип результата. Имена параметров локальны внутри power, это значит, что они скрыты для любой другой функции, так что остальные подпрограммы могут свободно пользоваться теми же именами для своих целей. Последнее утверждение справедливо также для переменных i и p: i в power и i в main не имеют между собой ничего общего.

Далее *параметром* мы будем называть переменную из списка параметров, заключенного в круглые скобки и заданного в определении функции, а *аргументом* - значение, используемое при обращении к функции. Иногда в том же смысле мы будем употреблять термины *формальный аргумент* и *фактический аргумент*.

Значение, вычисляемое функцией power, возвращается в main с помощью инструкции return. За словом return может следовать любое выражение:

return *выражение;*

Функция не обязательно возвращает какое-нибудь значение. Инструкция return без выражения только передает управление в ту программу, которая ее вызвала, не передавая ей никакого результирующего значения. То же самое происходит, если в процессе вычислений мы выходим на конец функции, обозначенный в тексте последней закрывающей фигурной скобкой. Возможна ситуация, когда вызывающая функция игнорирует возвращаемый ей результат.

Вы, вероятно, обратили внимание на инструкцию return в конце main. Поскольку main есть функция, как и любая другая она может вернуть результирующее значение тому, кто ее вызвал, - фактически в ту среду, из которой была запущена программа. Обычно возвращается нулевое значение, что говорит о нормальном завершении выполнения. Ненулевое значение сигнализирует о необычном или ошибочном завершении. До сих пор ради простоты мы опускали return в main, но с этого момента будем задавать return как напоминание о том, что программы должны сообщать о состоянии своего завершения в операционную систему.

Объявление

```
int power(int m, int n);
```

стоящее непосредственно перед main, сообщает, что функция power ожидает двух аргументов типа int и возвращает результат типа int. Это объявление, называемое *прототипом функции,* должно быть согласовано с определением и всеми вызовами power. Если определение функции или вызов не соответствует своему прототипу, это ошибка.

Имена параметров не требуют согласования. Фактически в прототипе они могут быть произвольными или вообще отсутствовать, т. е. прототип можно было бы записать и так:

```
int power(int, int);
```

Однако удачно подобранные имена поясняют программу, и мы будем часто этим пользоваться.

Историческая справка. Самые большие отличия ANSI-Си от более ранних версий языка как раз и заключаются в способах объявления и определения функций. В первой версии Си функцию power требовалось задавать в следующем виде:

```
/* power: возводит base в n-ю степень; n >= 0 */
/*        (версия в старом стиле языка Си) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Здесь имена параметров перечислены в круглых скобках, а их типы заданы перед первой открывающей фигурной скобкой. В случае отсутствия

указания о типе параметра, считается, что он имеет тип int. (Тело функции не претерпело изменений.)

Описание power в начале программы согласно первой версии Си должно было бы выглядеть следующим образом:

```
int power();
```

Нельзя было задавать список параметров, и поэтому компилятор не имел возможности проверить правильность обращений к power. Так как при отсутствии объявления power предполагалось, что функция возвращает значение типа int, то в данном случае объявление целиком можно было бы опустить.

Новый синтаксис для прототипов функций облегчает компилятору обнаружение ошибок в количестве аргументов и их типах. Старый синтаксис объявления и определения функции все еще допускается стандартом ANSI, по крайней мере на переходный период, но если ваш компилятор поддерживает новый синтаксис, мы настоятельно рекомендуем пользоваться только им.

Упражнение 1.15. Перепишите программу преобразования температур, выделив само преобразование в отдельную функцию.

1.8. Аргументы. Вызов по значению

Одно свойство функций в Си, вероятно, будет в новинку для программистов, которые уже пользовались другими языками, в частности Фортраном. В Си все аргументы функции передаются "по значению". Это следует понимать так, что вызываемой функции посылаются значения ее аргументов во временных переменных, а не сами аргументы. Такой способ передачи аргументов несколько отличается от "вызова по ссылке" в Фортране и спецификации var при параметре в Паскале, которые позволяют подпрограмме иметь доступ к самим аргументам, а не к их локальным копиям.

Главное отличие заключается в том, что в Си вызываемая функция не может непосредственно изменить переменную вызывающей функции: она может изменить только ее частную, временную копию.

Однако вызов по значению следует отнести к достоинствам языка, а не к его недостаткам. Благодаря этому свойству обычно удается написать более компактную программу, содержащую меньшее число посторонних переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные вызванной подпрограммы. В качестве примера приведем еще одну версию функции power, в которой как раз использовано это свойство.

```
/* power: возводит base в n-ю степень: n >= 0; версия 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр n выступает здесь в роли временной переменной, в которой циклом for в убывающем порядке ведется счет числа шагов до тех пор, пока ее значение не станет нулем. При этом отпадает надобность в дополнительной переменной i для счетчика цикла. Что бы мы ни делали с n внутри power, это не окажет никакого влияния на сам аргумент, копия которого была передана функции power при ее вызове.

При желании можно сделать так, чтобы функция смогла изменить переменную в вызывающей программе. Для этого последняя должна передать *адрес* подлежащей изменению переменной (*указатель* на переменную), а в вызываемой функции следует объявить соответствующий параметр как указатель и организовать через него косвенный доступ к этой переменной. Все, что касается указателей, мы рассмотрим в главе 5.

Механизм передачи массива в качестве аргумента несколько иной. Когда аргументом является имя массива, то функции передается значение, которое является адресом начала этого массива; никакие элементы массива не копируются. С помощью индексирования относительно полученного значения функция имеет доступ к любому элементу массива. Разговор об этом пойдет в следующем параграфе.

1.9. Символьные массивы

Самый распространенный вид массива в Си - массив символов. Чтобы Проиллюстрировать использование символьных массивов и работающих с ними функций, напишем программу, которая читает набор текстовых строк и печатает самую длинную из них. Ее схема достаточно проста:

```
while (есть ли еще строка?)
    if (данная строка длиннее самой длинной из предыдущих)
        запомнить ее
        запомнить ее длину
напечатать самую длинную строку
```

Из схемы видно, что программа естественным образом распадается на части. Одна из них получает новую строку, другая проверяет ее, третья запоминает, а остальные управляют процессом вычислений.

Поскольку процесс четко распадается на части, хорошо бы так и перевести его на Си. Поэтому сначала напишем отдельную функцию getline для получения очередной строки. Мы попытаемся сделать эту функцию полезной и для других применений. Как минимум getline должна сигнализировать о возможном конце файла, а еще лучше, если она будет выдавать длину строки — или нуль в случае исчерпания файла. Нуль годится для признака конца файла, поскольку не бывает строк нулевой длины, даже строка, содержащая только один символ новой строки, имеет длину 1.

Когда мы обнаружили строку более длинную, чем самая длинная из всех предыдущих, то нам надо будет где-то ее запомнить. Здесь напрашивается вторая функция, copy, которая умеет копировать новую строку в надежное место.

Наконец, нам необходима главная программа, которая бы управляла функциями getline и copy. Вот как выглядит наша программа в целом:

```
#include<stdio.h>
#define MAXLINE 1000 /* максимальный размер вводимой строки */

int getline(char line[], int MAXLINE);
void copy(char to[], char from[]);

/* печать самой длинной строки */
main()
{
  int len;       /* длина текущей строки */
  int max;       /* длина максимальной из просмотренных строк */
  char line[MAXLINE];          /* текущая строка */
  char longest[MAXLINE];       /* самая длинная строка */

  max = 0;
  while ((len = getline(line, MAXLINE)) > 0)
    if (len > max) {
      max = len;
      copy(longest, line);
    }
  if (max > 0)                 /* была ли хоть одна строка? */
    printf("%s", longest);
  return 0;
}
```

```
/* getline: читает строку в s, возвращает длину */
int getline(char s[], int lim)
{
    int c, i;
    for (i = 0; i < lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
        s[i] = c;
    if (c == 'n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: копирует т 'from' в 'to'; to достаточно большой */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Мы предполагаем, что функции getline и copy, описанные в начале программы, находятся в том же файле, что и main.

Функции main и getline взаимодействуют между собой через пару аргументов и возвращаемое значение. В getline аргументы определяются строкой

```
int getline(char s[], int lim)
```

Как мы видим, ее первый аргумент s есть массив, а второй, lim, имеет тип int. Задание размера массива в определении имеет целью резервирование памяти. В самой getline задавать длину массива s нет необходимости, так как его размер указан в main. Чтобы вернуть значение вызывающей программе, getline использует return точно так же, как это делает функция power. В приведенной строке также сообщается, что getline возвращает значение типа int, но так как при отсутствии указания о типе подразумевается int, то перед getline слово int можно опустить.

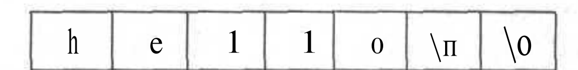
Одни функции возвращают результирующее значение, другие (такие как copy) нужны только для того, чтобы произвести какие-то действия,

не выдавая никакого значения. На месте типа результата в copy стоит void. Это явное указание на то, что никакого значения данная функция не возвращает.

Функция getline в конец создаваемого ею массива помещает символ '\0' *(null-символ,* кодируемый нулевым байтом), чтобы пометить конец строки символов. То же соглашение относительно окончания нулем соблюдается и в случае строковой константы вроде

```
"hello\n"
```

В данном случае для него формируется массив из символов этой строки с '\0' в конце.



Спецификация %s в формате printf предполагает, что соответствующий ей аргумент - строка символов, оформленная указанным выше образом. Функция copy в своей работе также опирается на тот факт, что читаемый ею аргумент заканчивается символом '\0', который она копирует наряду с остальными символами. (Все сказанное предполагает, что '\0' не встречается внутри обычного текста.)

Попутно стоит заметить, что при работе даже с такой маленькой программой мы сталкиваемся с некоторыми конструктивными трудностями. Например, что должна делать main, если встретится строка, превышающая допустимый размер? Функция getline работает надежно: если массив полон, она прекращает пересылку, даже если символа новой строки не обнаружила. Получив от getline длину строки и увидев, что она совпадает с MAXLINE, главная программа main могла бы "отловить" этот особый случай и справиться с ним. В интересах краткости описание этого случая мы здесь опускаем.

Пользователи getline не могут заранее узнать, сколь длинными будут вводимые строки, поэтому getline делает проверки на переполнение. А вот пользователям функции copy размеры копируемых строк известны (или они могут их узнать), поэтому дополнительный контроль здесь не нужен.

Упражнение 1.16. Перепишите main предыдущей программы так, чтобы она могла печатать самую длинную строку без каких-либо ограничений на ее размер.

Упражнение 1.17. Напишите программу печати всех вводимых строк, содержащих более 80 символов.

Упражнение 1.18. Напишите программу, которая будет в каждой вводимой строке заменять стоящие подряд символы пробелов и табуляций на один пробел и удалять пустые строки.

Упражнение 1.19. Напишите функцию reverse(s), размещающую символы в строке s в обратном порядке. Примените ее при написании программы, которая каждую вводимую строку располагает в обратном порядке.

1.10. Внешние переменные и область видимости

Переменные line, longest и прочие принадлежат только функции main, или, как говорят, локальны в ней. Поскольку они объявлены внутри main, никакие другие функции прямо к ним обращаться не могут. То же верно и применительно к переменным других функций. Например, i в getline не имеет никакого отношения к i в copy. Каждая локальная переменная функции возникает только в момент обращения к этой функции и исчезает после выхода из нее. Вот почему такие переменные, следуя терминологии других языков, называют *автоматическими.* (В главе 4 обсуждается класс памяти static, который позволяет локальным переменным сохранять свои значения в промежутках между вызовами.)

Так как автоматические переменные образуются и исчезают одновременно с входом в функцию и выходом из нее, они не сохраняют своих значений от вызова к вызову и должны устанавливаться заново при каждом новом обращении к функции. Если этого не делать, они будут содержать "мусор".

В качестве альтернативы автоматическим переменным можно определить *внешние* переменные, к которым разрешается обращаться по их именам из любой функции. (Этот механизм аналогичен области COMMON в Фортране и определениям переменных в самом внешнем блоке в Паскале.) Так как внешние переменные доступны повсеместно, их можно использовать вместо аргументов для связи между функциями по данным. Кроме того, поскольку внешние переменные существуют постоянно, а не возникают и исчезают на период выполнения функции, свои значения они сохраняют и после возврата из функций, их установивших.

Внешняя переменная должна быть *определена,* причем только один раз, вне текста любой функции; в этом случае ей будет выделена память. Она

должна быть *объявлена* во всех функциях, которые хотят ею пользоваться. Объявление содержит сведения о типе переменной. Объявление может быть явным, в виде инструкции extern, или неявным, когда нужная информация получается из контекста. Чтобы конкретизировать сказанное, перепишем программу печати самой длинной строки с использованием line, longest и max в качестве внешних переменных. Это потребует изменений в вызовах, объявлениях и телах всех трех функций.

```
#include <stdio.h>

tfdefine MAXLINE 1000    /* максимальный размер вводимой строки */

int max;      /* длина максимальной из просмотренных строк */
char line[MAXLINE];      /* текущая строка */
char longest[MAXLINE];   /* самая длинная строка */

int getline(void);
void copy(void);

/* печать самой длинной строки; специализированная версия */
main ()
{
  int len;
  extern int max;
  extern char longest[];

  max = 0;
  while ((len= getline()) > 0)
     if (len > max) {
       max = len;
       copy();
     }
  if (max > 0)          /* была хотя бы одна строка */
     printf("%s", longest);
  return 0;
}

/* getline: специализированная версия */
int getline(void)
{
  int c, i;
  extern char line[];
```

```
    for (i = 0; i < MAXLINE-1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i]=c;
    if(c == '\n') {
        line[i]= c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: специализированная версия */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Внешние переменные для main, getline и copy определяются в начале нашего примера, где им присваивается тип и выделяется память. Определения внешних переменных синтаксически ничем не отличаются от определения локальных переменных, но поскольку они расположены вне функций, эти переменные считаются внешними. Чтобы функция могла пользоваться внешней переменной, ей нужно прежде всего сообщить имя соответствующей переменной. Это можно сделать, например, задав объявление extern, которое по виду отличается от объявления внешней переменной только тем, что оно начинается с ключевого слова extern.

В некоторых случаях объявление extern можно опустить. Если определение внешней переменной в исходном файле расположено выше функции, где она используется, то в объявлении extern нет необходимости. Таким образом, в main, getline и copy объявления extern избыточны. Обычно определения внешних переменных располагают в начале исходного файла, и все объявления exte rn для них опускают.

Если же программа расположена в нескольких исходных файлах и внешняя переменная определена в *файле 1,* а используется в *файле2* и *файле3,* то объявления extern в *файле2* и *файле3* обязательны, поскольку необходимо указать, что во всех трех файлах функции обращаются к одной и той же внешней переменной. На практике обычно удобно собрать все

объявления внешних переменных и функций в отдельный файл, называемый *заголовочным (header-файлом),* и помещать его с помощью #include в начало каждого исходного файла. В именах header-файлов по общей договоренности используется суффикс .h. В этих файлах, в частности в <stdio.h>, описываются также функции стандартной библиотеки. Более подробно о заголовочных файлах говорится в главе 4, а применительно к стандартной библиотеке - в главе 7 и приложении В.

Так как специализированные версии getline и copy не имеют аргументов, на первый взгляд кажется, что логично их прототипы задать в виде getline() и copy(). Но из соображений совместимости со старыми Си-программами стандарт рассматривает пустой список как сигнал к тому, чтобы выключить все проверки на соответствие аргументов. Поэтому, когда нужно сохранить контроль и явно указать отсутствие аргументов, следует пользоваться словом void. Мы вернемся к этой проблеме в главе 4.

Заметим, что по отношению к внешним переменным в этом параграфе мы очень аккуратно используем понятия *определение* и *объявление.* "Определение" располагается в месте, где переменная создается и ей отводится память; "объявление" помещается там, где фиксируется природа переменной, но никакой памяти для нее не отводится.

Следует отметить тенденцию все переменные делать внешними. Дело в том, что, как может показаться на первый взгляд, это приводит к упрощению связей - ведь списки аргументов становятся короче, а переменные доступны везде, где они нужны; однако они оказываются доступными и там, где не нужны. Так что чрезмерный упор на внешние переменные чреват большими опасностями - он приводит к созданию программ, в которых связи по данным не очевидны, поскольку переменные могут неожиданным и даже таинственным способом изменяться. Кроме того, такая программа с трудом поддается модификациям. Вторая версия программы поиска самой длинной строки хуже, чем первая, отчасти по этим причинам, а отчасти из-за нарушения общности двух полезных функций, вызванного тем, что в них вписаны имена конкретных переменных, с которыми они оперируют.

Итак, мы рассмотрели то, что можно было бы назвать ядром Си. Описанных "кирпичиков" достаточно, чтобы создавать полезные программы значительных размеров, и было бы чудесно, если бы вы, прервав чтение, посвятили этому какое-то время. В следующих упражнениях мы предлагаем вам создать несколько более сложные программы, чем рассмотренные выше.

Упражнение 1.20. Напишите программу detab, заменяющую символы табуляции во вводимом тексте нужным числом пробелов (до следующего "стопа" табуляции). Предполагается, что "стопы" табуляции расставлены

на фиксированном расстоянии друг от друга, скажем, через n позиций. Как лучше задавать n – в виде значения переменной или в виде именованной константы?

Упражнение 1.21. Напишите программу entab, заменяющую строки из пробелов минимальным числом табуляций и пробелов таким образом, чтобы вид напечатанного текста не изменился. Используйте те же "стопы" табуляции, что и в detab. В случае, когда для выхода на очередной "стоп" годится один пробел, что лучше - пробел или табуляция?

Упражнение 1.22. Напишите программу, печатающую символы входного потока так, чтобы строки текста не выходили правее n-й позиции. Это значит, что каждая строка, длина которой превышает n, должна печататься с переносом на следующие строки. Место переноса следует "искать" после последнего символа, отличного от символа-разделителя, расположенного левее л-й позиции. Позаботьтесь о том, чтобы ваша программа вела себя разумно в случае очень длинных строк, а также когда до n-й позиции не встречается ни одного символа пробела или табуляции.

Упражнение 1.23. Напишите программу, убирающую все комментарии из любой Си-программы. Не забудьте должным образом обработать строки символов и строковые константы. Комментарии в Си не могут быть вложены друг в друга.

Упражнение 1.24. Напишите программу, проверяющую Си-программы на элементарные синтаксические ошибки вроде несбалансированности скобок всех видов. Не забудьте о кавычках (одиночных и двойных), эскейп-последовательностях (\. . .) и комментариях. (Это сложная программа, если писать ее для общего случая.)

Глава 2

Типы, операторы и выражения

Переменные и константы являются основными объектами данных, с которыми имеет дело программа. Переменные перечисляются в объявлениях, где устанавливаются их типы и, возможно, начальные значения. Операции определяют действия, которые совершаются с этими переменными. Выражения комбинируют переменные и константы для получения новых значений, Тип объекта определяет множество значений, которые этот объект может принимать, и операций, которые над ними могут выполняться. Названные "кирпичики" и будут предметом обсуждения в этой главе.

Стандартом ANSI было утверждено значительное число небольших изменений и добавлений к основным типам и выражениям. Любой целочисленный тип теперь может быть со знаком, signed, и без знака, unsigned. Предусмотрен способ записи беззнаковых констант и шестнадцатеричных символьных констант. Операции с плавающей точкой допускаются теперь и с одинарной точностью. Введен тип long double, обеспечивающий повышенную точность. Строковые константы конкатенируются ("склеиваются") теперь во время компиляции. Частью языка стали перечисления (enum), формализующие для типа установку диапазона значений. Объекты для защиты их от каких-либо изменений разрешено помечать как const. В связи с введением новых типов расширены правила автоматического преобразования из одного арифметического типа в другой.

2.1. Имена переменных

Хотя мы ничего не говорили об этом в главе 1, но существуют некоторые ограничения на задание имен переменных и именованных констант.

Имена составляются из букв и цифр; первым символом должна быть буква. Символ подчеркивания "_" считается буквой; его иногда удобно использовать, чтобы улучшить восприятие длинных имен переменных. Не начинайте имена переменных с подчеркивания, так как многие переменные библиотечных программ начинаются именно с этого знака. Большие (прописные) и малые (строчные) буквы различаются, так что x и X — это два разных имени. Обычно в программах на Си малыми буквами набирают переменные, а большими - именованные константы.

Для внутренних имен значимыми являются первые 31 символ. Для имен функций и внешних переменных число значимых символов может быть меньше 31, так как эти имена обрабатываются ассемблерами и загрузчиками и языком не контролируются. Уникальность внешних имен гарантируется только в пределах 6 символов, набранных безразлично в каком регистре. Ключевые слова if, else, int, float и т. д. зарезервированы, и их нельзя использовать в качестве имен переменных. Все они набираются на нижнем регистре (т. е. малыми буквами).

Разумно давать переменным осмысленные имена в соответствии с их назначением, причем такие, чтобы их было трудно спутать друг с другом. Мы предпочитаем короткие имена для локальных переменных, особенно для счетчиков циклов, и более длинные для внешних переменных.

2.2. Типы и размеры данных

В Си существует всего лишь несколько базовых типов:

- char - единичный байт, который может содержать один символ из допустимого символьного набора;
- int - целое, обычно отображающее естественное представление целых в машине;
- float - число с плавающей точкой одинарной точности;
- double - число с плавающей точкой двойной точности.

Имеется также несколько квалификаторов, которые можно использовать вместе с указанными базовыми типами. Например, квалификаторы short (короткий) и long (длинный) применяются к целым:

```
short int sh;
long int counter;
```

В таких объявлениях слово int можно опускать, что обычно и делается.

Если только не возникает противоречий со здравым смыслом, short int и long int должны быть разной длины, а int соответствовать естественному размеру целых на данной машине. Чаще всего для представления

целого, описанного с квалификатором short, отводится 16 битов, с квалификатором long - 32 бита, а значению типа int - или 16, или 32 бита. Разработчики компилятора вправе сами выбирать подходящие размеры, сообразуясь с характеристиками своего компьютера и соблюдая следующие ограничения: значения типов short и int представляются по крайней мере 16 битами; типа long - по крайней мере 32 битами; размер short не больше размера int, который в свою очередь не больше размера long.

Квалификаторы signed (со знаком) или unsigned (без знака) можно применять к типу char и любому целочисленному типу. Значения unsigned всегда положительны или равны нулю и подчиняются законам арифметики по модулю 2^n, где *n* - количество битов в представлении типа. Так, если значению char отводится 8 битов, то unsigned char имеет значения в диапазоне от 0 до 255, а signed char - от -128 до 127 (в машине с двоичным дополнительным кодом). Являются ли значения типа просто char знаковыми или беззнаковыми, зависит от реализации, но в любом случае коды печатаемых символов положительны.

Тип long double предназначен для арифметики с плавающей точкой повышенной точности. Как и в случае целых, размеры объектов с плавающей точкой зависят от реализации; float, double и long double могут представляться одним размером, а могут - двумя или тремя разными размерами.

Именованные константы для всех размеров вместе с другими характеристиками машины и компилятора содержатся в стандартных заголовочных файлах <limits.h> и <float.h> (см. приложение В).

Упражнение 2.1. Напишите программу, которая будет выдавать диапазоны значений типов char, short, int и long, описанных как signed и как unsigned, с помощью печати соответствующих значений из стандартных заголовочных файлов и путем прямого вычисления. Определите диапазоны чисел с плавающей точкой различных типов. Вычислить эти диапазоны сложнее.

2.3. Константы

Целая константа, например 1234, имеет тип int. Константа типа long завершается буквой l или L, например 123456789L; слишком большое целое, которое невозможно представить как int, будет представлено как long. Беззнаковые константы заканчиваются буквой u или U, а окончание ul или UL говорит о том, что тип константы - unsigned long.

Константы с плавающей точкой имеют десятичную точку (123.4), или экспоненциальную часть (1e-2), или же и то и другое. Если у них нет окон-

чания, считается, что они принадлежат к типу double. Окончание f или F указывает на тип float, а 1 или L - на тип long double.

Целое значение помимо десятичного может иметь восьмеричное или шестнадцатеричное представление. Если константа начинается с нуля, то она представлена в восьмеричном виде, если с Ox или с OX, то — в шестнадцатеричном. Например, десятичное целое 31 можно записать как 037 или как OX1F. Записи восьмеричной и шестнадцатеричной констант могут завершаться буквой L (для указания на тип long) и U (если нужно показать, что константа беззнаковая). Например, константа OXFUL имеет значение 15 и тип unsigned long.

Символьная константа есть целое, записанное в виде символа, обрамленного одиночными кавычками, например 'x'. Значением символьной константы является числовой код символа из набора символов на данной машине. Например, символьная константа '0' в кодировке ASCII имеет значение 48, которое никакого отношения к числовому значению О не имеет. Когда мы пишем '0', а не какое-то значение (например 48), зависящее от способа кодировки, мы делаем программу независимой от частного значения кода, к тому же она и легче читается. Символьные константы могут участвовать в операциях над числами точно так же, как и любые другие целые, хотя чаще они используются для сравнения с другими символами.

Некоторые символы в символьных и строковых константах записываются с помощью эскейп-последовательностей, например \n (символ новой строки); такие последовательности изображаются двумя символами, но обозначают один. Кроме того, произвольный восьмеричный код можно задать в виде

'*ooo*'

где *ooo* - одна, две или три восьмеричные цифры (0...7) или

'\x*hh*'

где *hh* - одна, две или более шестнадцатеричные цифры (0... 9, a...f, A...F). Таким образом, мы могли бы написать

```
#define VTAB '013' /* вертикальная табуляция в ASCII */
#define BELL '\007' /* звонок В ASCII */
```

или в шестнадцатеричном виде:

```
#define VTAB '\xb' /* вертикальная табуляция в ASCII */
#define BELL '\x7' /* звонок в ASCII */
```

Полный набор эскейп-последовательностей таков:

\a	сигнал-звонок	\\	обратная наклонная черта

\b	возврат-на-шаг (забой)	\?	знак вопроса
\f	перевод-страницы	\'	одиночная кавычка
\n	новая-строка	\"	двойная кавычка
\r	возврат-каретки	\ooo	восьмеричный код
\t	горизонтальная-табуляция	\xhh	шестнадцатеричный код
\v	вертикальная-табуляция		

Символьная константа '\0' - это символ с нулевым значением, так называемый символ null. Вместо просто 0 часто используют запись '\0', чтобы подчеркнуть символьную природу выражения, хотя и в том и другом случае запись обозначает нуль.

Константные выражения – это выражения, оперирующие только с константами. Такие выражения вычисляются во время компиляции, а не во время выполнения, и поэтому их можно использовать в любом месте, где допустимы константы, как, например, в

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

или в

```
#define LEAP 1 /* in leap years - в високосные годы */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Строковая константа, или *строковый литерал,* - это нуль или более символов, заключенных в двойные кавычки, как, например,

```
"Я строковая константа"
```

или

```
"" /* пустая строка */
```

Кавычки не входят в строку, а служат только ее ограничителями. Так же, как и в символьные константы, в строки можно включать эскейп-последовательности; \", например, представляет собой двойную кавычку. Строковые константы можно конкатенировать ("склеивать") во время компиляции; например, запись двух строк

```
"Здравствуй," " мир!"
```

эквивалентна записи одной следующей строки:

```
"Здравствуй, мир!"
```

Указанное свойство позволяет разбивать длинные строки на части и располагать эти части на отдельных строчках.

Фактически строковая константа - это массив символов. Во внутреннем представлении строки в конце обязательно присутствует нулевой символ '\0', поэтому памяти для строки требуется на один байт больше, чем число символов, расположенных между двойными кавычками. Это означает, что на длину задаваемой строки нет ограничения, но чтобы определить ее длину, требуется просмотреть всю строку. Функция strlen(s) вычисляет длину строки s без учета завершающего ее символа '\0'. Ниже приводится наша версия этой функции:

```
/* strlen: возвращает длину строки s */
int strlen(char s[])
{
    int i;

    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Функция strlen и некоторые другие, применяемые к строкам, описаны в стандартном заголовочном файле <string.h>.

Будьте внимательны и помните, что символьная константа и строка, содержащая один символ, не одно и то же: 'x' не то же самое, что "x". Запись 'x' обозначает целое значение, равное коду буквы x из стандартного символьного набора, а запись "x" - массив символов, который содержит один символ (букву x) и '\0'.

В Си имеется еще один вид константы - *константа перечисления*. Перечисление – это список целых констант, как, например, в

```
enum boolean { NO, YES };
```

Первое имя в enum[1] имеет значение 0, следующее - 1, и т. д. (если для значений констант не было явных спецификаций). Если не все значения специфицированы, то они продолжают прогрессию, начиная от последнего специфицированного значения, как в следующих двух примерах:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
             /* FEB есть 2, MAR есть 3 и т.д. */
```

[1] От английского слова *enumeration* — перечисление. — *Примеч. ред.*

Имена в различных перечислениях должны отличаться друг от друга. Значения внутри одного перечисления могут совпадать.

Средство enum обеспечивает удобный способ присвоить константам имена, причем в отличие от #define значения констант при этом способе могут генерироваться автоматически. Хотя разрешается объявлять переменные типа enum, однако компилятор не обязан контролировать, входят ли присваиваемые этим переменным значения в их тип. Но сама возможность такой проверки часто делает enum лучше, чем #define. Кроме того, отладчик получает возможность печатать значения переменных типа enum в символьном виде.

2.4. Объявления

Все переменные должны быть объявлены раньше, чем будут использоваться, при этом некоторые объявления могут быть получены неявно - из контекста. Объявление специфицирует тип и содержит список из одной или нескольких переменных этого типа, как, например, в

```
int lower, upper, step;
char c, line [1000];
```

Переменные можно распределять по объявлениям произвольным образом, так что указанные выше списки можно записать и в следующем виде:

```
int   lower;
int   upper;
int   step;
char  c;
char  line[1000];
```

Последняя форма записи занимает больше места, тем не менее она лучше, поскольку позволяет добавлять к каждому объявлению комментарий. Кроме того, она более удобна для последующих модификаций.

В своем объявлении переменная может быть инициализирована, как, например:

```
char   esc = '\\';
int    i = 0;
int    limit = MAXLINE+1;
float  eps = 1.0e-5;
```

Инициализация неавтоматической переменной осуществляется только один раз - перед тем, как программа начнет выполняться, при этом начальное значение должно быть константным выражением. Явно ини-

циализируемая автоматическая переменная получает начальное значение каждый раз при входе в функцию или блок, ее начальным значением может быть любое выражение. Внешние и статические переменные по умолчанию получают нулевые значения. Автоматические переменные, явным образом не инициализированные, содержат неопределенные значения ("мусор").

К любой переменной в объявлении может быть применен квалификатор const для указания того, что ее значение далее не будет изменяться.

```
const double e = 2.71828182845905;
const char msg[] = "предупреждение: ";
```

Применительно к массиву квалификатор const указывает на то, что ни один из его элементов не будет меняться. Указание const можно также применять к аргументу-массиву, чтобы сообщить, что функция не изменяет этот массив:

```
int strlen(const char[] );
```

Реакция на попытку изменить переменную, помеченную квалификатором const, зависит от реализации компилятора.

2.5. Арифметические операторы

Бинарными (т. е. с двумя операндами) арифметическими операторами являются +, -, *, /, а также оператор деления по модулю %. Деление целых сопровождается отбрасыванием дробной части, какой бы она ни была. Выражение

```
x % y
```

дает остаток от деления x на y и, следовательно, нуль, если x делится на y нацело. Например, год является високосным, если он делится на 4, но не делится на 100. Кроме того, год является високосным, если он делится на 400. Следовательно,

```
if ((year % 4 == 0 && year % 100 != O !i year % 400 == 0)
    printf("%d високосный год\n", year);
else
    printf("%d невисокосный год\n", year);
```

Оператор % к операндам типов float и double не применяется. В какую сторону (в сторону увеличения или уменьшения числа) будет усечена дробная часть при выполнении / и каким будет знак результата операции % с отрицательными операндами, зависит от машины.

Бинарные операторы + и - имеют одинаковый приоритет, который ниже приоритета операторов *, / и %, который в свою очередь ниже приоритета унарных операторов + и -. Арифметические операции одного приоритетного уровня выполняются слева направо.

В конце этой главы (параграф 2.12) приводится таблица 2.1, в которой представлены приоритеты всех операторов и очередность их выполнения.

2.6. Операторы отношения и логические операторы

Операторами отношения являются

```
>   >=   <   <=
```

Все они имеют одинаковый приоритет. Сразу за ними идет приоритет операторов сравнения на равенство:

```
==   !=
```

Операторы отношения имеют более низкий приоритет, чем арифметические, поэтому выражение вроде i < lim-1 будет выполняться так же, как i < (lim-1), т. е. как мы и ожидаем.

Более интересны логические операторы && и ||. Выражения, между которыми стоят операторы && или ||, вычисляются слева направо. Вычисление прекращается, как только становится известна истинность или ложность результата. Многие Си-программы опираются на это свойство, как, например, цикл из функции getline, которую мы приводили в главе 1:

```
for (i = 0; i < lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
    s[i] = c;
```

Прежде чем читать очередной символ, нужно проверить, есть ли для него место в массиве s, иначе говоря, сначала *необходимо* проверить соблюдение условия i < lim-1. Если это условие не выполняется, мы не должны продолжать вычисление, в частности читать следующий символ. Так же было бы неправильным сравнивать c и EOF до обращения к getchar; следовательно, и вызов getchar, и присваивание должны выполняться перед указанной проверкой.

Приоритет оператора && выше, чем таковой оператора ||, однако их приоритеты ниже, чем приоритет операторов отношения и равенства. Из сказанного следует, что выражение вида

```
i < lim-1 && (c = getchar()) != '\n' && c != EOF
```

не нуждается в дополнительных скобках. Но, так как приоритет != выше,

чем приоритет присваивания, в

```
(c = getchar()) != '\n'
```

скобки необходимы, чтобы сначала выполнить присваивание, а затем сравнение с '\n'.

По определению численным результатом вычисления выражения отношения или логического выражения является 1, если оно истинно, и О, если оно ложно.

Унарный оператор ! преобразует ненулевой операнд в 0, а нуль в 1. Обычно оператор ! используют в конструкциях вида

```
if (!valid)
```

что эквивалентно

```
if (valid == 0)
```

Трудно сказать, какая из форм записи лучше. Конструкция вида !valid хорошо читается ("если не правильно"), но в более сложных выражениях может оказаться, что ее не так-то легко понять.

Упражнение 2.2. Напишите цикл, эквивалентный приведенному выше for-циклу, не пользуясь операторами && и ||.

2.7. Преобразования типов

Если операнды оператора принадлежат к разным типам, то они приводятся к некоторому общему типу. Приведение выполняется в соответствии с небольшим числом правил. Обычно автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном, как, например, преобразование целого в число с плавающей точкой в выражении вроде f + i. Выражения, не имеющие смысла, например число с плавающей точкой в роли индекса, не допускаются. Выражения, в которых могла бы теряться информация (скажем, при присваивании длинных целых переменным более коротких типов или при присваивании значений с плавающей точкой целым переменным), могут повлечь за собой предупреждение, но они допустимы.

Значения типа char - это просто малые целые, и их можно свободно использовать в арифметических выражениях, что значительно облегчает всевозможные манипуляции с символами. В качестве примера приведем простенькую реализацию функции atoi, преобразующей последовательность цифр в ее числовой эквивалент.

```
/* atoi: преобразование s в целое */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Как мы уже говорили в главе 1, выражение

```
s[i] -'0'
```

дает числовое значение символа, хранящегося в s[i], так как значения '0', '1' и пр. образуют непрерывную возрастающую последовательность.

Другой пример приведения char к int связан с функцией lower, которая одиночный символ из набора ASCII, если он является заглавной буквой, превращает в строчную. Если же символ не является заглавной буквой, lower его не изменяет.

```
/* lower: преобразование c в строчную; только для ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

В случае ASCII эта программа будет работать правильно, потому что между одноименными буквами верхнего и нижнего регистров - одинаковое расстояние (если их рассматривать как числовые значения). Кроме того, латинский алфавит — плотный, т. е. между буквами A и Z расположены только буквы. Для набора EBCDIC последнее условие не выполняется, и поэтому наша программа в этом случае будет преобразовывать не только буквы.

Стандартный заголовочный файл <ctype.h>, описанный в приложении B, определяет семейство функций, которые позволяют проверять и преобразовывать символы независимо от символьного набора. Например, функция tolower(c) возвращает букву c в коде нижнего регистра, если она была в коде верхнего регистра, поэтому tolower - универсальная замена функции lower, рассмотренной выше. Аналогично проверку

```
c >= '0' && c <= '9'
```

можно заменить на

```
isdigit(c)
```

Далее мы будем пользоваться функциями из <ctype.h>.

Существует одна тонкость, касающаяся преобразования символов в целые числа: язык не определяет, являются ли переменные типа char знаковыми или беззнаковыми. При преобразовании char в int может ли когда-нибудь получиться отрицательное целое? На машинах с разной архитектурой ответы могут отличаться. На некоторых машинах значение типа char с единичным старшим битом будет превращено в отрицательное целое (посредством "распространения знака"). На других - преобразование char в int осуществляется добавлением нулей слева, и, таким образом, получаемое значение всегда положительно.

Гарантируется, что любой символ из стандартного набора печатаемых символов никогда не будет отрицательным числом, поэтому в выражениях такие символы всегда являются положительными операндами. Но произвольный восьмибитовый код в переменной типа char на одних машинах может быть отрицательным числом, а на других - положительным. Для совместимости переменные типа char, в которых хранятся несимвольные данные, следует специфицировать явно как signed или unsigned.

Отношения вроде i > j и логические выражения, перемежаемые операторами && и ||, определяют выражение-условие, которое имеет значение 1, если оно истинно, и 0, если ложно. Так, присваивание

```
d = c >= '0' && c <= '9'
```

установит d в значение 1, если c есть цифра, и 0 в противном случае. Однако функции, подобные isdigit, в качестве истины могут выдавать любое ненулевое значение. В местах проверок внутри if, while, for и пр. "истина" просто означает "не нуль".

Неявные арифметические преобразования, как правило, осуществляются естественным образом. В общем случае, когда оператор вроде + или * с двумя операндами (бинарный оператор) имеет разнотипные операнды, прежде чем операция начнет выполняться, "низший" тип *повышается* до "высшего". Результат будет иметь высший тип. В параграфе 6 приложения А правила преобразования сформулированы точно. Если же в выражении нет беззнаковых операндов, можно удовлетвориться следующим набором неформальных правил:

- Если какой-либо из операндов принадлежит типу long double, то и другой приводится к long double.
- В противном случае, если какой-либо из операндов принадлежит типу double, то и другой приводится к double.

- В противном случае, если какой-либо из операндов принадлежит типу float, то и другой приводится к float.
- В противном случае операнды типов char и short приводятся к int.
- И наконец, если один из операндов типа long, то и другой приводится к long.

Заметим, что операнды типа float не приводятся автоматически к типу double; в этом данная версия языка отличается от первоначальной. Вообще говоря, математические функции, аналогичные собранным в библиотеке <math.h>, базируются на вычислениях с двойной точностью. В основном float используется для экономии памяти на больших массивах и не так часто - для ускорения счета на тех машинах, где арифметика с двойной точностью слишком дорога с точки зрения расхода времени и памяти.

Правила преобразования усложняются с появлением операндов типа unsigned. Проблема в том, что сравнения знаковых и беззнаковых значений зависят от размеров целочисленных типов, которые на разных машинах могут отличаться. Предположим, что значение типа int занимает 16 битов, а значение типа long - 32 бита. Тогда -1L < 1U, поскольку 1U принадлежит типу unsigned int и повышается до типа signed long. Но -1L > 1UL, так как -1L повышается до типа unsigned long и воспринимается как большое положительное число.

Преобразования имеют место и при присвоениях: значение правой части присвоения приводится к типу левой части, который и является типом результата.

Тип char превращается в int путем распространения знака или другим описанным выше способом.

Тип long int преобразуются в short int или в значения типа char путем отбрасывания старших разрядов. Так, в

```
int i;
char c;

i = c;
c = i;
```

значение c не изменится. Это справедливо независимо от того, распространяется знак при переводе char в int или нет. Однако, если изменить очередность присваиваний, возможна потеря информации.

Если x принадлежит типу float, a i - типу int, то и x = i, и i = x вызовут преобразования, причем перевод float в int сопровождается отбрасыванием дробной части. Если double переводится во float, то значение либо округляется, либо обрезается; это зависит от реализации.

Так как аргумент в вызове функции есть выражение, при передаче его функции также возможно преобразование типа. При отсутствии прото-

типа функции аргументы типа char и short переводятся в int, а float - в double. Вот почему мы объявляли аргументы типа int или double даже тогда, когда в вызове функции использовали аргументы типа char или float.

И наконец, для любого выражения можно явно ("насильно") указать преобразование его типа, используя унарный оператор, называемый *приведением*. Конструкция вида

(*имя-типа*) *выражение*

приводит *выражение к* указанному в скобках типу по перечисленным выше правилам. Смысл операции приведения можно представить себе так: *выражение* как бы присваивается некоторой переменной указанного типа, и эта переменная используется вместо всей конструкции. Например, библиотечная функция sqrt рассчитана на аргумент типа double и выдает чепуху, если ей подсунуть что-нибудь другое (sqrt описана в <math.h>). Поэтому, если n имеет целочисленный тип, мы можем написать

```
sqrt((double) n)
```

и перед тем, как значение n будет передано функции, оно будет переведено в double. Заметим, что операция приведения всего лишь вырабатывает *значение* n указанного типа, но саму переменную n не затрагивает. Приоритет оператора приведения столь же высок, как и любого унарного оператора, что зафиксировано в таблице, помещенной в конце этой главы.

В том случае, когда аргументы описаны в прототипе функции, как тому и следует быть, при вызове функции нужное преобразование выполняется автоматически. Так, при наличии прототипа функции sqrt:

```
double sqrt(double);
```

перед обращением к sqrt в присваивании

```
root2 = sqrt(2);
```

целое 2 будет переведено в значение double 2.0 автоматически без явного указания операции приведения.

Операцию приведения проиллюстрируем на переносимой версии генератора псевдослучайных чисел и функции, инициализирующей "семя". И генератор, и функция входят в стандартную библиотеку.

```
unsigned long int next = 1;

/* rand: возвращает псевдослучайное целое 0...32767 */
int rand(void)
{
  next = next * 1103515245 + 12345;
```

```
    return (unsigned int)(next/65536) % 32768;
}

/* srand: устанавливает "семя" для rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Упражнение 2.3. Напишите функцию htol(s), которая преобразует последовательность шестнадцатеричных цифр, начинающуюся с 0x или 0X, в соответствующее целое. Шестнадцатеричными цифрами являются символы 0...9, a...f, A...F.

2.8. Операторы инкремента и декремента

В Си есть два необычных оператора, предназначенных для увеличения и уменьшения переменных. Оператор инкремента ++ добавляет 1 к своему операнду, а оператор декремента — вычитает 1. Мы уже неоднократно использовали ++ для наращивания значения переменных, как, например, в

```
if (c == '\n')
    ++nl;
```

Необычность операторов ++ и — в том, что их можно использовать и как префиксные (помещая перед переменной: ++n), и как постфиксные (помещая после переменной: n++) операторы. В обоих случаях значение n увеличивается на 1, но выражение ++n увеличивает n до того, как его значение будет использовано, а n++ - *после* того. Предположим, что n содержит 5, тогда

```
x = n++;
```

установит x в значение 5, а

```
x = ++n;
```

установит x в значение 6. И в том и другом случае n станет равным 6. Операторы инкремента и декремента можно применять только к переменным. Выражения вроде (i+j)++ недопустимы.

Если требуется только увеличить или уменьшить значение переменной (но не получить ее значение), как например

```
if (c == '\n')
    nl++;
```

то безразлично, какой оператор выбрать - префиксный или постфиксный. Но существуют ситуации, когда требуется оператор вполне определенного типа. Например, рассмотрим функцию squeeze(s,c), которая удаляет из строки s все символы, совпадающие с c:

```
/* squeeze: удаляет все с из s */
void squeeze(char s[], int c)
{
  int i, j;
  for (i = j = 0; s[i] != '\0' i++)
    if (s[i] != c)
      s[j++] = s[i];
  s[i] = '\0';
}
```

Каждый раз, когда встречается символ, отличный от c, он копируется в текущую j-ю позицию, и только после этого переменная j увеличивается на 1, подготавливаясь таким образом к приему следующего символа. Это в точности совпадает со следующими действиями:

```
if (s[i] != c) {
  s[j] = s[i];
  j++;
}
```

Другой пример - функция getline, которая нам известна по главе 1. Приведенную там запись

```
if (c == '\n') {
  s[i] = c;
  ++i;
}
```

можно переписать более компактно:

```
if (c == '\n')
  s[i++] = c;
```

В качестве третьего примера рассмотрим стандартную функцию strcat(s,t), которая строку t помещает в конец строки s. Предполагается, что в s достаточно места, чтобы разместить там суммарную строку. Мы написали strcat так, что она не возвращает никакого результата. На самом деле библиотечная strcat возвращает указатель на результирующую строку.

```
/* strcat: помещает t в конец s; s достаточно велика */
void strcat (char s[], char t[])
```

```
{
    int i,j;

    i = j = 0;
    while (s[i] != '\0')             /* находим конец s */
        i++;
    while ((s[i++] = t[j++]) != '\0'  /* копируем t */
        ;
}
```

При копировании очередного символа из t в s постфиксный оператор ++ применяется и к i, и к j, чтобы на каждом шаге цикла переменные i и j правильно отслеживали позиции перемещаемого символа.

Упражнение 2.4. Напишите версию функции squeeze(s1,s2), которая удаляет из s1 все символы, встречающиеся в *строке* s2.

Упражнение 2.5. Напишите функцию any(s1,s2), которая возвращает либо ту позицию в s1, где стоит первый символ, совпавший с любым из символов в s2, либо -1 (если ни один символ из s1 не совпадает с символами из s2). (Стандартная библиотечная функция strpbrk делает то же самое, но выдает не номер позиции символа, а указатель на символ.)

2.9. Побитовые операторы

В Си имеются шесть операторов для манипулирования с битами. Их можно применять только к целочисленным операндам, т. е. к операндам типов char, short, int и long, знаковым и беззнаковым.

&	- побитовое И.
i	- побитовое ИЛИ.
^	- побитовое исключающее ИЛИ.
<<	- сдвиг влево.
>>	- сдвиг вправо.
~	- побитовое отрицание (унарный).

Оператор & (побитовое И) часто используется для обнуления некоторой группы разрядов. Например

```
n = n & 0177;
```

обнуляет в n все разряды, кроме младших семи.

Оператор ! (побитовое ИЛИ) применяют для установки разрядов; так,

```
x = x ! SET_ON;
```

устанавливает единицы в тех разрядах x, которым соответствуют единицы в SET_ON.

Оператор ^(побитовое исключающее ИЛИ) в каждом разряде установит 1, если соответствующие разряды операндов имеют различные значения, и 0, когда они совпадают.

Поразрядные операторы & и ! следует отличать от логических операторов && и !!, которые при вычислении слева направо дают значение истинности. Например, если x равно 1, а y равно 2, то x & y даст нуль, а x && y - единицу.

Операторы << и >> сдвигают влево или вправо свой левый операнд на число битовых позиций, задаваемое правым операндом, который должен быть неотрицательным. Так, x << 2 сдвигает значение x влево на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению x на 4. Сдвиг вправо беззнаковой величины всегда сопровождается заполнением освобождающихся разрядов нулями. Сдвиг вправо знаковой величины на одних машинах происходит с распространением знака ("арифметический сдвиг"), на других - с заполнением освобождающихся разрядов нулями ("логический сдвиг").

Унарный оператор ~ поразрядно "обращает" целое т. е. превращает каждый единичный бит в нулевой и наоборот. Например

```
x = x & ~077
```

обнуляет в x последние 6 разрядов. Заметим, что запись x & ~077 не зависит от длины слова, и, следовательно, она лучше, чем x & 0177700, поскольку последняя подразумевает, что x занимает 16 битов. Не зависимая от машины форма записи -077 не потребует дополнительных затрат при счете, так как ~077 - константное выражение, которое будет вычислено во время компиляции.

Для иллюстрации некоторых побитовых операций рассмотрим функцию getbits(x, p, n), которая формирует поле в n битов, вырезанных из x, начиная с позиции p, прижимая его к правому краю. Предполагается, что 0-й бит - крайний правый бит, а n и p - осмысленные положительные числа. Например, getbits(x, 4, 3) вернет в качестве результата 4, 3 и 2-й биты значения x, прижимая их к правому краю. Вот эта функция:

```
/* getbits: получает n бит, начиная с p-й позиции */
unsigned getbits(unsigned x, int p, int n)
{
  return (x >> (p+1-n)) & ~(~0 << n);
}
```

Выражение x >> (p+1-n) сдвигает нужное нам поле к правому краю. Константа ~0 состоит из одних единиц, и ее сдвиг влево на п бит (~0 << п) приведет к тому, что правый край этой константы займут п нулевых разрядов. Еще одна операция побитовой инверсии ~ позволяет получить справа п единиц.

Упражнение 2.6. Напишите функцию setbits(x, p, n, y), возвращающую значение x, в котором п битов, начиная с p-й позиции, заменены на п правых разрядов из y (остальные биты не изменяются).

Упражнение 2.7. Напишите функцию invert(x, p, п), возвращающую значение x с инвертированными п битами, начиная *с* позиции p (остальные биты не изменяются).

Упражнение 2.8. Напишите функцию rightrot(x, n), которая циклически сдвигает x вправо на п разрядов.

2.10. Операторы и выражения присваивания

Выражение

    i = i + 2

в котором стоящая слева переменная повторяется и справа, можно написать в сжатом виде:

    i += 2

Оператор +=, как и =, называется *оператором присваивания*.

Большинству бинарных операторов (аналогичных + и имеющих левый и правый операнды) соответствуют операторы присваивания *op*=, где *op* - один из операторов

    +   *   /   %   <<   >>   &   ^   ;

Если $выр_1$ и $выр_2$ - выражения, то

$выр_1$ *op*= $выр_2$

эквивалентно

$выр_1 = (выр_1)$ *op* $(выр_2)$

с той лишь разницей, что $выр_1$ вычисляется только один раз. Обратите внимание на скобки вокруг $выр_2$:

```
x *= y + 1
```

эквивалентно

```
x = x * (y + 1)
```

но не

```
x = x * y + 1
```

В качестве примера приведем функцию bitcount, подсчитывающую число единичных битов в своем аргументе целочисленного типа.

```
/* bitcount: подсчет единиц в x */
int bitcount( unsigned x)
{
  int b;
  for (b = 0; x != 0; x >>= 1)
    if (x & 01)
      b++;
  return b;
}
```

Независимо от машины, на которой будет работать эта программа, объявление аргумента x как unsigned гарантирует, что при правом сдвиге освобождающиеся биты будут заполняться нулями, а не знаковым битом.

Помимо краткости операторы присваивания обладают тем преимуществом, что они более соответствуют тому, как человек мыслит. Мы говорим "прибавить 2 к i" или "увеличить i на 2", а не "взять i, добавить 2 и затем вернуть результат в i", так что выражение i += 2 лучше, чем i = i + 2. Кроме того, в сложных выражениях вроде

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

благодаря оператору присваивания += запись становится более легкой для понимания, так как читателю при такой записи не потребуется старательно сравнивать два длинных выражения, совпадают ли они, или выяснять, почему они не совпадают. Следует иметь в виду и то, что подобные операторы присваивания могут помочь компилятору сгенерировать более эффективный код.

Мы уже видели, что присваивание вырабатывает значение и может применяться внутри выражения; вот самый расхожий пример:

```
while ((c = getchar()) != EOF).
  ...
```

В выражениях встречаются и другие операторы присваивания (+=, -= и т. д.), хотя и реже.

Типом и значением любого выражения присваивания являются тип и значение его левого операнда после завершения присваивания.

Упражнение 2.9. Применительно к числам, в представлении которых использован дополнительный код, выражение x &= (x-1) уничтожает самую правую 1 в x. Объясните, почему. Используйте это наблюдение при написании более быстрого варианта функции bitcount.

2.11. Условные выражения

Инструкции

```
if (a > Ь)
   z = a;
else
   г = b;
```

пересылают в z большее из двух значений a и b. *Условное выражение,* написанное с помощью тернарного (т. е. имеющего три операнда) оператора "? :", представляет собой другой способ записи этой и подобных ей конструкций. В выражении

выр$_1$ *1 выр*$_2$: *выр*$_3$

первым вычисляется выражение *выр*$_1$. Если его значение не нуль (истина), то вычисляется выражение *выр*$_2$, и значение этого выражения становится значением всего условного выражения. В противном случае вычисляется выражение *выр*$_3$, и его значение становится значением условного выражения. Следует отметить, что из выражений *выр*$_2$ и *выр*$_3$ вычисляется только одно из них. Таким образом, чтобы установить в z большее из a и b, можно написать

```
z = (a > b) ? a : b;        /* z = max(a, b) */
```

Следует заметить, что условное выражение и в самом деле является выражением, и его можно использовать в любом месте, где допускается выражение. Если *выр*$_2$ и *выр*$_3$ принадлежат разным типам, то тип результата определяется правилами преобразования, о которых шла речь в этой главе ранее. Например, если f имеет тип float, а n — тип int, то типом выражения

```
(л > 0) ? f : п
```

будет float вне зависимости от того, положительно значение п или нет.

Заключать в скобки первое выражение в условном выражении не обязательно, так как приоритет ?: очень низкий (более низкий приоритет имеет только присваивание), однако мы рекомендуем всегда это делать, поскольку благодаря обрамляющим скобкам условие в выражении лучше воспринимается.

Условное выражение часто позволяет сократить программу. В качестве примера приведем цикл, обеспечивающий печать n элементов массива по 10 на каждой строке с одним пробелом между колонками; каждая строка цикла, включая последнюю, заканчивается символом новой строки:

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10 == 9 || i == n-1) ? '\n' : ' ');
```

Символ новой строки посылается после каждого десятого и после n-го элемента. За всеми другими элементами следует пробел. Эта программа выглядит довольно замысловато, зато она более компактна, чем эквивалентная программа с использованием if-else. Вот еще один хороший пример[1]:

```
printf("Вы имеете %d элемент%s.\n", n, (n%10 == 1 && n%100 != 11) ?
  " " : ((n%100 < 10 || n%100 > 20) && n%10 >= 2 && n%10 <= 4) ?
  "а" : "ов");
```

Упражнение 2.10. Напишите функцию lower, которая переводит большие буквы в малые, используя условное выражение (а не конструкцию if-else).

2.12. Приоритет и очередность вычислений

В таблице 2.1 показаны приоритеты и очередность вычислений всех операторов, включая и те, которые мы еще не рассматривали. Операторы, перечисленные на одной строке, имеют одинаковый приоритет; строки упорядочены по убыванию приоритетов; так, например, *, / и % имеют одинаковый приоритет, который выше, чем приоритет бинарных + и -. "Оператор" () относится к вызову функции. Операторы -> и . (точка) обеспечивают доступ к элементам структур; о них пойдет речь в главе 6, там же будет рассмотрен и оператор sizeof (размер объекта). Операторы * (косвенное обращение по указателю) и & (получение адреса объекта) обсуждаются в главе 5. Оператор "запятая" будет рассмотрен в главе 3.

Таблица 2.1. Приоритеты и очередность вычислений операторов

Операторы	Выполняются
() [] -> .	слева направо
! ~ ++ -- + - * & *(тип)* sizeof	справа налево
* / %	слева направо

[1] Этот пример, учитывающий русскую грамматику, отличается от авторского оригинала. - *Примеч. ред.*

Продолжение табл. 2.1

Операторы	Выполняются
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
!	слева направо
&&	слева направо
||	слева направо
?:	справа налево
= += -= *= /= %= &= ^= |= <<= >>=	справа налево
,	слева направо

Примечание. Унарные операторы +, -, * и & имеют более высокий приоритет, чем те же бинарные операторы.

Заметим, что приоритеты побитовых операторов &, ^ и ! ниже, чем приоритет == и !=, из-за чего в побитовых проверках, таких как

```
if ((x & MASK) == 0) ...
```

чтобы получить правильный результат, приходится использовать скобки.

Си подобно многим языкам не фиксирует очередность вычисления операндов оператора (за исключением &&, ||, ?: и ,). Например, в инструкции вида

```
x = f() + g();
```

f может быть вычислена раньше g или наоборот. Из этого следует, что если одна из функций изменяет значение переменной, от которой зависит другая функция, то помещаемый в x результат может зависеть от очередности вычислений. Чтобы обеспечить нужную последовательность вычислений, промежуточные результаты можно запоминать во временных переменных.

Очередность вычисления аргументов функции также не определена, поэтому на разных компиляторах

```
printf("%d %d\n", ++n, power(2, n));        /* НЕВЕРНО */
```

может давать несовпадающие результаты. Результат вызова функции зависит от того, когда компилятор сгенерирует команды увеличения n — до или после обращения к power. Чтобы обезопасить себя от возможного побочного эффекта, достаточно написать

```
++n;
printf("%d %d\n", n, power(2, n));
```

Обращения к функциям, вложенные присвоения, инкрементные и декрементные операторы дают "побочный эффект", проявляющийся в том, что при вычислении выражения значения некоторых переменных изменяются. В любом выражении с побочным эффектом может быть скрыта трудно просматриваемая зависимость результата выражения от очередности изменения значений переменных, входящих в выражение. В такой, например, типично неприятной ситуации

```
a[i] = i++;
```

возникает вопрос: массив a индексируется старым или измененным значением i? Компиляторы могут по-разному генерировать программу, что проявится в интерпретации данной записи. Стандарт сознательно устроен так, что большинство подобных вопросов оставлено на усмотрение компиляторов, так как лучший порядок вычислений определяется архитектурой машины. Стандартом только гарантируется, что все побочные эффекты при вычислении аргументов проявятся перед входом в функцию. Правда, в примере с `printf` это нам не поможет.

Мораль такова: писать программы, зависящие от очередности вычислений, - плохая практика, какой бы язык вы ни использовали. Естественно, надо знать, чего следует избегать, но если вы не знаете, *как* образуются побочные эффекты на разных машинах, то лучше и не рассчитывать выиграть на особенностях частной реализации.

Глава 3

Управление

Порядок, в котором выполняются вычисления, определяется инструкциями управления. Мы уже встречались с наиболее распространенными управляющими конструкциями такого рода в предыдущих примерах; здесь мы завершим их список и более точно определим рассмотренные ранее.

3.1. Инструкции и блоки

Выражение, скажем x = 0, или i++, или printf (...), становится *инструкцией,* если в конце его поставить точку *с* запятой, например:

```
x = 0;
i++;
printf(...);
```

В Си точка с запятой является заключающим символом инструкции, а не разделителем, как в языке Паскаль.

Фигурные скобки { и } используются для объединения объявлений и инструкций в *составную инструкцию,* или *блок,* чтобы с точки зрения синтаксиса эта новая конструкция воспринималась как одна инструкция. Фигурные скобки, обрамляющие группу инструкций, образующих тело функции, - это один пример; второй пример - это скобки, объединяющие инструкции, помещенные после if, else, while или for. (Переменные могут быть объявлены внутри *любого* блока, об этом разговор пойдет в главе 4.) После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится.

3.2. Конструкция if-else

Инструкция if-else используется для принятия решения. Формально ее синтаксисом является:

```
if  (выражение)
   инструкция₁
else
   инструкция₂
```

причем else-часть может и отсутствовать. Сначала вычисляется *выражение,* и, если оно истинно (т. е. отлично от нуля), выполняется *инструкция*$_1$. Если *выражение* ложно (т. е. его значение равно нулю) и существует else-часть, то выполняется *инструкция*$_2$.

Так как if просто проверяет числовое значение выражения, условие иногда можно записывать в сокращенном виде. Так, запись

```
if  (выражение)
```

короче, чем

```
if (выражение != 0)
```

Иногда такие сокращения естественны и ясны, в других случаях, наоборот, затрудняют понимание программы.

Отсутствие else-части в одной из вложенных друг в друга if-конструкций может привести к неоднозначному толкованию записи. Эту неоднозначность разрешают тем, что else связывают с ближайшим if, у которого нет своего else. Например, в

```
if (n > 0)
   if (a > Ь)
      2 = a;
   else
      2 = b;
```

else относится к внутреннему if, что мы и показали с помощью отступов. Если нам требуется иная интерпретация, необходимо должным образом расставить фигурные скобки:

```
if (n > 0) {
   if (a > b)
      2 = a;
}
else
   2 = b;
```

Ниже приводится пример ситуации, когда неоднозначность особенно опасна:

```
if (n >= 0)
   for (i = 0; i < n; i++)
```

```
        if (s[i] > 0) {
            printf ("...");
            return i;
        }
    else                  /* НЕВЕРНО */
        printf("ошибка - отрицательное n\n");
```

С помощью отступов мы недвусмысленно показали, что нам нужно, однако компилятор не воспримет эту информацию и отнесет else к внутреннему `if`. Искать такого рода ошибки особенно тяжело. Здесь уместен следующий совет: вложенные `if` обрамляйте фигурными скобками.

Кстати, обратите внимание на точку с запятой после z = a в

```
if (a > b)
    z = a;
else
    z = b;
```

Здесь она обязательна, поскольку по правилам грамматики за if должна следовать инструкция, а выражение-инструкция вроде z = a; всегда заканчивается точкой с запятой.

3.3. Конструкция `else-if`

Конструкция

if *(выражение)*
    инструкция
else if *(выражение)*
    инструкция
else if *(выражение)*
    инструкция
else if *(выражение)*
    инструкция
else
    инструкция

встречается так часто, что о ней стоит поговорить особо. Приведенная последовательность инструкций if - самый общий способ описания многоступенчатого принятия решения. *Выражения* вычисляются по порядку; как только встречается *выражение* со значением "истина", выполняется соответствующая ему *инструкция*; на этом последовательность проверок завершается. Здесь под словом *инструкция* имеется в виду либо одна

инструкция, либо группа инструкций в фигурных скобках.

Последняя else-часть срабатывает, если не выполняются все предыдущие условия. Иногда в последней части не требуется производить никаких действий, в этом случае фрагмент

```
else
    инструкция
```

можно опустить или использовать для фиксации ошибочной ("невозможной") ситуации.

В качестве иллюстрации трехпутевого ветвления рассмотрим функцию бинарного поиска значения x в массиве v. Предполагается, что элементы v упорядочены по возрастанию. Функция выдает положение x в v (число в пределах от 0 до n-1), если x там встречается, и -1, если его нет.

При бинарном поиске значение x сначала сравнивается с элементом, занимающим серединное положение в массиве v. Если x меньше, чем это значение, то областью поиска становится "верхняя" половина массива v, в противном случае - "нижняя". В любом случае следующий шаг - это сравнение с серединным элементом отобранной половины. Процесс "уполовинивания" диапазона продолжается до тех пор, пока либо не будет найдено значение, либо не станет пустым диапазон поиска. Запишем функцию бинарного поиска:

```
/* binsearch: найти x в v[0] <= v[1] <= . . . <= v[n-1] */
int binsearch(int x, int v[], int n)

{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else            /* совпадение найдено */
            return mid;
    }
    return -1;      /* совпадения нет */
}
```

Основное действие, выполняемое на каждом шаге поиска, - сравнение значения x (меньше, больше или равно) с элементом v[mid]; это сравнение естественно поручить конструкции else-if.

Упражнение 3.1. В нашей программе бинарного поиска внутри цикла осуществляются две проверки, хотя могла быть только одна (при увеличении числа проверок вне цикла). Напишите программу, предусмотрев в ней одну проверку внутри цикла. Оцените разницу во времени выполнения.

3.4. Переключатель switch

Инструкция switch используется для выбора одного из многих путей. Она проверяет, совпадает ли значение выражения с одним из значений, входящих в некоторое множество целых *констант,* и выполняет соответствующую этому значению ветвь программы:

```
switch (выражение) {
   case конст-выр: инструкции
   case конст-выр: инструкции
   default: инструкции
}
```

Каждая ветвь case помечена одной или несколькими целочисленными константами или же константными выражениями. Вычисления начинаются с той ветви case, в которой константа совпадает со значением выражения. Константы всех ветвей case должны отличаться друг от друга. Если выяснилось, что ни одна из констант не подходит, то выполняется ветвь, помеченная словом default, если таковая имеется, в противном случае ничего не делается. Ветви case и default можно располагать в любом порядке.

В главе 1 мы написали программу, подсчитывающую число вхождений в текст каждой цифры, символов-разделителей (пробелов, табуляций и новых строк) и всех остальных символов. В ней мы использовали последовательность if... else if... else. Теперь приведем вариант этой программы с переключателем switch:

```
tfinclude <stdio. h>

main() /* подсчет цифр, символов-разделителей и прочих символов */
{
   int c, i, nwhite, nother, ndigit[10];
```

```
    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i]=0;
    while ((c= getchar()) != EOF) {
        switch (c) {
        case '0' : case '1' : case '2' : case '3' : case '4' :
        case '5' : case '6' : case '7' : case '8' : case '9' :
            ndigit[c - '0']++;
            break;
        case ' ' :
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf ("цифр =");
    for (i= 0; i < 10; i++)
        printf (" %d", ndigit[i]);
    printf(", символов-разделителей = %d, прочих = %d\n",
        nwhite, nother);
    return 0;
}
```

Инструкция break вызывает немедленный выход из переключателя switch. Поскольку выбор ветви case реализуется как переход на метку, то после выполнения одной ветви case, если ничего не предпринять, программа *провалится вниз* на следующую ветвь. Инструкции break и return — наиболее распространенные средства выхода из переключателя. Инструкция break используется также для принудительного выхода из циклов while, for и do-while (мы еще поговорим об этом чуть позже).

"Сквозное" выполнение ветвей case вызывает смешанные чувства. С одной стороны, это хорошо, поскольку позволяет несколько ветвей case объединить в одну, как мы и поступили с цифрами в нашем примере. Но с другой - это означает, что в конце почти каждой ветви придется ставить break, чтобы избежать перехода к следующей. Последовательный проход по ветвям - вещь ненадежная, это чревато ошибками, особенно при изменении программы. За исключением случая с несколькими метками для одного вычисления, старайтесь по возможности реже пользоваться сквозным

проходом, но если уж вы его применяете, обязательно комментируйте эти особые места.

Добрый вам совет: даже в конце последней ветви (после default в нашем примере) помещайте инструкцию break, хотя с точки зрения логики в ней нет никакой необходимости. Но эта маленькая предосторожность спасет вас, когда однажды вам потребуется добавить в конец еще одну ветвь case.

Упражнение 3.2. Напишите функцию escape(s, t), которая при копировании текста из t в s преобразует такие символы, как новая строка и табуляция в "видимые последовательности символов" (вроде \л и \t). Используйте инструкцию switch. Напишите функцию, выполняющую обратное преобразование эскейп-последовательностей в настоящие символы.

3.5. Циклы while и for

Мы уже встречались с циклами while и for. В цикле

```
while (выражение)
    инструкция
```

вычисляется *выражение*. Если его значение отлично от нуля, то выполняется *инструкция*, и вычисление выражения повторяется. Этот цикл продолжается до тех пор, пока выражение не станет равным нулю, после чего вычисления продолжатся с точки, расположенной сразу за *инструкцией*.

Инструкция for

```
for (выр1; выр2; выр3)
    инструкция
```

эквивалентна конструкции

```
выр1;
while (выр2) {
    инструкция
    выр3;
}
```

если не считать отличий в поведении инструкции continue, речь о которой пойдет в параграфе 3. 7.

С точки зрения грамматики три компоненты цикла for представляют собой произвольные выражения, но чаще *выр*$_1$ и *выр*$_3$ — это присваивания или вызовы функций, а *выр*$_2$ - выражение отношения. Любое из этих трех

выражений может отсутствовать, но точку с запятой опускать нельзя. При отсутствии *выр*$_1$ или *выр*$_3$ считается, что их просто нет в конструкции цикла; при отсутствии *выр*$_2$ предполагается, что его значение как бы всегда истинно. Например,

```
for (;;) {
    ...
}
```

есть "бесконечный" цикл, выполнение которого, вероятно, прерывается каким-то другим способом, например с помощью инструкций break или return.

Какой цикл выбрать: while или for - это дело вкуса. Так, в

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;                   /* обойти символы-разделители */
```

нет ни инициализации, ни пересчета параметра, поэтому здесь больше подходит while.

Там, где есть простая инициализация и пошаговое увеличение значения некоторой переменной, больше подходит цикл for, так как в этом цикле организующая его часть сосредоточена в начале записи. Например, начало цикла, обрабатывающего первые n элементов массива, имеет следующий вид:

```
for (i = 0; i < n; i++)
    ...
```

Это похоже на DO-циклы в Фортране и for-циклы в Паскале. Сходство, однако, не вполне точное, так как в Си индекс и его предельное значение могут изменяться внутри цикла, и значение индекса i после выхода из цикла всегда определено. Поскольку три компоненты цикла могут быть произвольными выражениями, организация for-циклов не ограничивается только случаем арифметической прогрессии. Однако включать в заголовок цикла вычисления, не имеющие отношения к инициализации и инкрементированию, считается плохим стилем. Заголовок лучше оставить только для операций управления циклом.

В качестве более внушительного примера приведем другую версию программы atoi, выполняющей преобразование строки в ее числовой эквивалент. Это более общая версия по сравнению с рассмотренной в главе 2, в том смысле, что она игнорирует левые символы-разделители (если они есть) и должным образом реагирует на знаки + и -, которые могут стоять перед цифрами. (В главе 4 будет рассмотрен вариант atof, который осуществляет подобное преобразование для чисел с плавающей точкой.)

Структура программы отражает вид вводимой информации:

игнорировать символы-разделители, если они есть
получить знак, если он есть
взять целую часть и преобразовать ее

На каждом шаге выполняется определенная часть работы и четко фиксируется ее результат, который затем используется на следующем шаге. Обработка данных заканчивается на первом же символе, который не может быть частью числа.

```
#include <ctype.h>

/* atoi: преобразование s в целое число; версия 2 */
int atoi(char s[])
{
    int i, n, sign;
                                /* игнорировать символы-разделители */
    for (i = 0; isspace(s[i]); i++)
        ;
    sign = (s[i] == '-') ? -1: 1;
    if (s[i] == '+' || s[i] == '-')     /* пропуск знака */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

Заметим, что в стандартной библиотеке имеется более совершенная функция преобразования строки в длинное целое (long int) - функция strtol (см. параграф 5 приложения В).

Преимущества, которые дает централизация управления циклом, становятся еще более очевидными, когда несколько циклов вложены друг в друга. Проиллюстрируем их на примере сортировки массива целых чисел методом Шелла, предложенным им в 1959 г. Основная идея этого алгоритма в том, что на ранних стадиях сравниваются далеко отстоящие друг от друга, а не соседние элементы, как в обычных перестановочных сортировках. Это приводит к быстрому устранению массовой неупорядоченности, благодаря чему на более поздней стадии остается меньше работы. Интервал между сравниваемыми элементами постепенно уменьшается до единицы, и в этот момент сортировка сводится к обычным перестановкам соседних элементов. Программа shellsort имеет следующий вид:

```
/* shellsort: сортируются v[0]... v[n-1] в возрастающем порядке */
void shellsort (int v[], int n)
```

```
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && v[j] > v[j + gap]; j -= gap) {
                temp = v[j];
                v[j] = v[j + gap];
                v[j + gap] = temp;
            }
}
```

Здесь использованы три вложенных друг в друга цикла. Внешний управляет интервалом gap между сравниваемыми элементами, сокращая его путем деления пополам от n/2 до нуля. Средний цикл перебирает элементы. Внутренний - сравнивает каждую пару элементов, отстоящих друг от друга на расстоянии gap, и переставляет элементы в неупорядоченных парах. Так как gap обязательно сведется к единице, все элементы в конечном счете будут упорядочены. Обратите внимание на то, что универсальность цикла for позволяет сделать внешний цикл по форме похожим на другие, хотя он и не является арифметической прогрессией.

Последний оператор Си - это "," (запятая), которую чаще всего используют в инструкции for. Пара выражений, разделенных запятой, вычисляется слева направо. Типом и значением результата являются тип и значение правого выражения, что позволяет в инструкции for в каждой из трех компонент иметь по нескольку выражений, например вести два индекса параллельно. Продемонстрируем это на примере функции reverse(s), которая "переворачивает" строку s, оставляя результат в той же строке s:

```
#include <string.h>

/* reverse: переворачивает строку s (результат в s) */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Запятые, разделяющие аргументы функции, переменные в объявлениях и пр. *не* являются операторами-запятыми и не обеспечивают вычислений слева направо.

Запятыми как операторами следует пользоваться умеренно. Более всего они уместны в конструкциях, которые тесно связаны друг с другом (как в for-цикле программы reverse), а также в макросах, в которых многоступенчатые вычисления должны быть выражены одним выражением. Запятой-оператором в программе reverse можно было бы воспользоваться и при обмене символами в проверяемых парах элементов строки, мысля этот обмен как одну отдельную операцию:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
   c = s[i], s[i] = s[j], s[j] = c;
```

Упражнение 3.3. Напишите функцию expand(s1, s2), заменяющую сокращенную запись наподобие a-z в строке s1 эквивалентной полной записью abc. . . xyz в s2. В s1 допускаются буквы (прописные и строчные) и цифры. Следует уметь справляться с такими случаями, как a-b-c, a-z0--9 и -a-b. Считайте знак - в начале или в конце s1 обычным символом минус.

3.6. Цикл do-while

Как мы говорили в главе 1, в циклах while и for проверка условия окончания цикла выполняется наверху. В Си имеется еще один вид цикла, do-while, в котором эта проверка в отличие от while и for делается внизу *после* каждого прохождения тела цикла, т. е. после того, как тело выполнится хотя бы один раз. Цикл do-while имеет следующий синтаксис:

```
do
   инструкция
while (выражение);
```

Сначала выполняется *инструкция,* затем вычисляется *выражение.* Если оно истинно, то *инструкция* выполняется снова и т. д. Когда выражение становится ложным, цикл заканчивает работу. Цикл do-while эквивалентен циклу repeat-until в Паскале с той лишь разницей, что в первом случае указывается условие продолжения цикла, а во втором — условие его окончания.

Опыт показывает, что цикл do-while используется гораздо реже, чем while и for. Тем не менее потребность в нем время от времени возникает, как, например, в функции itoa (обратной по отношению к atoi), преоб-

разующей число в строку символов. Выполнить такое преобразование оказалось несколько более сложным делом, чем ожидалось, поскольку простые алгоритмы генерируют цифры в обратном порядке. Мы остановились на варианте, в котором сначала формируется обратная последовательность цифр, а затем она реверсируется.

```
/* itoa: преобразование n в строку s */
void itoa(int n, char s[])
{
  int i, sign;

  if ((sign = n) < 0)        /* сохраняем знак */
    n = -n;                  /* делаем n положительным */
  i = 0;
  do {   /* генерируем цифры в обратном порядке */
    s[i++] = n % 10 + '0'; /* следующая цифра */
  } while ((n /= 10) > 0);  /* исключить ее */
  if (sign < 0)
      s[i++] = '-';
  s[i] = '\0';
  reverse(s);
}
```

Конструкция do-while здесь необходима или по крайней мере удобна, поскольку в s посылается хотя бы один символ, даже если n равно нулю. В теле цикла одну инструкцию мы выделили фигурными скобками (хотя они и избыточны), чтобы неискушенный читатель не принял по ошибке слово while за *начало* цикла while.

Упражнение 3.4. При условии, что для представления чисел используется дополнительный код, наша версия itoa не справляется с самым большим по модулю отрицательным числом, значение которого равняется $-(2^{n-1})$, где *n* - размер слова. Объясните, чем это вызвано. Модифицируйте программу таким образом, чтобы она давала правильное значение указанного числа независимо от машины, на которой выполняется.

Упражнение 3.5. Напишите функцию itob(n,s,b), которая переводит целое n в строку s, представляющую число по основанию b. В частности, itob(n,s, 16) помещает в s текст числа n в шестнадцатеричном виде.

Упражнение 3.6. Напишите версию itoa с дополнительным третьим аргументом, задающим минимальную ширину поля. При необходимости преобразованное число должно слева дополняться пробелами.

3.7. Инструкции break и continue

Иногда бывает удобно выйти из цикла не по результату проверки, осуществляемой в начале или в конце цикла, а каким-то другим способом. Такую возможность для циклов for, while и do-while, а также для переключателя switch предоставляет инструкция break. Эта инструкция вызывает немедленный выход из самого внутреннего из объемлющих ее циклов или переключателей.

Следующая функция, trim, удаляет из строки завершающие пробелы, табуляции, символы новой строки; break используется в ней для выхода из цикла по первому обнаруженному справа символу, отличному от названных.

```
/* trim:  удаляет завершающие пробелы, табуляции
          и новые строки */
int trim(char s[])
{
  int n;

  for (n = strlen(s)-1; n >= 0; n--)
    if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
      break;
  s[n+1] = '\0';
  return n;
}
```

С помощью функции strlen можно получить длину строки. Цикл for просматривает его в обратном порядке, начиная с конца, до тех пор, пока не встретится символ, отличный от пробела, табуляции и новой строки. Цикл прерывается, как только такой символ обнаружится или n станет отрицательным (т. е. вся строка будет просмотрена). Убедитесь, что функция ведет себя правильно и в случаях, когда строка пуста или состоит только из символов-разделителей.

Инструкция continue в чем-то похожа на break, но применяется гораздо реже. Она вынуждает ближайший объемлющий ее цикл (for, while или do-while) начать следующий шаг итерации. Для while и do-while это означает немедленный переход к проверке условия, а для for - к приращению шага. Инструкцию continue можно применять только к циклам, но не к switch. Внутри переключателя switch, расположенного в цикле, она вызовет переход к следующей итерации этого цикла.

Вот фрагмент программы, обрабатывающий только неотрицательные элементы массива a (отрицательные пропускаются).

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)       /* пропуск отрицательных элементов */
        continue;
    ...                 /* обработка положительных элементов */
}
```

К инструкции continue часто прибегают тогда, когда оставшаяся часть цикла сложна, а замена условия в нем на противоположное и введение еще одного уровня приводят к слишком большому числу уровней вложенности.

3.8. Инструкция goto и метки

В Си имеются порицаемая многими инструкция goto и метки для перехода на них. Строго говоря, в этой инструкции нет никакой необходимости, и на практике почти всегда легко без нее обойтись. До сих пор в нашей книге мы не использовали goto.

Однако существуют случаи, в которых goto может пригодиться. Наиболее типична ситуация, когда нужно прервать обработку в некоторой глубоко вложенной структуре и выйти сразу из двух или большего числа вложенных циклов. Инструкция break здесь не поможет, так как она обеспечит выход только из самого внутреннего цикла. В качестве примера рассмотрим следующую конструкцию:

```
    for (...)
        for (...) {
            ...
            if (disaster)       /* если бедствие */
                goto error;     /* уйти на ошибку */
        }
    ...
error:                          /* обработка ошибки */
    ликвидировать беспорядок
```

Такая организация программы удобна, если подпрограмма обработки ошибочной ситуации не тривиальна и ошибка может встретиться в нескольких местах.

Метка имеет вид обычного имени переменной, за которым следует двоеточие. На метку можно перейти с помощью goto из любого места данной функции, т. е. метка видима на протяжении всей функции.

В качестве еще одного примера рассмотрим такую задачу: определить, есть ли в массивах a и b совпадающие элементы. Один из возможных вариантов ее реализации имеет следующий вид:

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto found;
    /* нет одинаковых элементов */
    ...
found:
    /* обнаружено совпадение: a[i] == b[j] */
    ...
```

Программу нахождения совпадающих элементов можно написать и без goto, правда, заплатив за это дополнительными проверками и еще одной переменной:

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
    if (found)
        /* обнаружено совпадение: a[i-1] == b[j-1] */
        ...
    else
        /* нет одинаковых элементов */
        ...
```

За исключением редких случаев, подобных только что приведенным, программы с применением goto, как правило, труднее для понимания и сопровождения, чем программы, решающие те же задачи без goto. Хотя мы и не догматики в данном вопросе, все же думается, что к goto следует прибегать крайне редко, если использовать эту инструкцию вообще.

Глава 4

Функции и структура программы

Функции разбивают большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками, а не начинать создание программы каждый раз "с нуля". В выбранных должным образом функциях "упрятаны" несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в нее изменений.

Язык проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на Си состоят из большого числа небольших функций, а не из немногих больших. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциями. Процесс загрузки здесь не рассматривается, поскольку он различен в разных системах.

Объявление и определение функции - это та область, где стандартом ANSI в язык внесены самые существенные изменения. Как мы видели в главе 1, в описании функции теперь разрешено задавать типы аргументов. Синтаксис определения функции также изменен, так что теперь объявления и определения функций соответствуют друг другу. Это позволяет компилятору обнаруживать намного больше ошибок, чем раньше. Кроме того, если типы аргументов соответствующим образом объявлены, то необходимые преобразования аргументов выполняются автоматически.

Стандарт вносит ясность в правила, определяющие области видимости имен; в частности, он требует, чтобы для каждого внешнего объекта было только одно определение. В нем обобщены средства инициализации: теперь можно инициализировать автоматические массивы и структуры.

Улучшен также препроцессор Си. Он включает более широкий набор директив условной компиляции, предоставляет возможность из макроаргументов генерировать строки в кавычках, а кроме того, содержит более совершенный механизм управления процессом макрорасширения.

4.1. Основные сведения о функциях

Начнем с того, что сконструируем программу, печатающую те строки вводимого текста, в которых содержится некоторый "образец", заданный в виде строки символов. (Эта программа представляет собой частный случай функции grep системы UNIX.) Рассмотрим пример: в результате поиска образца "ould" в строках текста

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits — and then
Re-mould it nearer to the Heart's Desire!
```

мы получим

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits — and then
Re-mould it nearer to the Heart's Desire!
```

Работа по поиску образца четко распадается на три этапа:

while (*существует еще строка)*
if (*строка содержит образец*)
напечатать ее

Хотя все три составляющие процесса поиска можно поместить в функцию main, все же лучше сохранить приведенную структуру и каждую ее часть реализовать в виде отдельной функции. Легче иметь дело с тремя небольшими частями, чем с одной большой, поскольку, если несущественные особенности реализации скрыты в функциях, вероятность их нежелательного воздействия друг на друга минимальна. Кроме того, оформленные в виде функций соответствующие части могут оказаться полезными и в других программах.

Конструкция "while *(существует еще строка)*" реализована в getline (см. главу 1), а фразу *"напечатать ее"* можно записать с помощью готовой функции printf. Таким образом, нам остается перевести на Си только то, что определяет, входит ли заданный образец в строку.

Чтобы решить эту задачу, мы напишем функцию strindex(s,t), которая указывает место (индекс) в строке s, где начинается строка t, или -1, если s не содержит t. Так как в Си нумерация элементов в массивах начинается с нуля, отрицательное число -1 подходит в качестве признака неудачного поиска. Если далее нам потребуется более сложное отождествление по образцу, мы просто заменим strindex на другую функцию, оставив при этом остальную часть программы без изменений. (Библиотечная функция strstr аналогична функции strindex и отличается от последней только тем, что возвращает не индекс, а указатель.)

После такого проектирования программы ее "деталировка" оказывается очевидной. Мы имеем представление о программе в целом и знаем, как взаимодействуют ее части. В нашей программе образец для поиска задается строкой-литералом, что снижает ее универсальность. В главе 5 мы еще вернемся к проблеме инициализации символьных массивов и покажем, как образец сделать параметром, устанавливаемым при запуске программы. Здесь приведена несколько измененная версия функции getline, и было бы поучительно сравнить ее с версией, рассмотренной в главе 1.

```
#include <stdio.h>
#define MAXLINE 1000        /* максимальный размер вводимой строки */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";  /* образец для поиска */
/* найти все строки, содержащие образец */
main()
{
  char line[MAXLINE];
  int found = 0;

  while (getline(line, MAXLINE) > 0)
    if (strindex(line, pattern) >= 0) {
      printf ("%s", line);
      found++;
    }
  return found;
}

/* getline: читает строку в s, возвращает длину */
int getline(char s[], int lim)
{
  int c, i;
```

```
    i = 0;
    while (-lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s [i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: вычисляет место t в s или выдает -1, если t нет в s */
int strindex (char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Определение любой функции имеет следующий вид:

тип-результата имя-функции (объявления аргументов)
{
объявления и инструкции
}

Отдельные части определения могут отсутствовать, как, например, в определении "минимальной" функции

```
dummy() {}
```

которая ничего не вычисляет и ничего не возвращает. Такая ничего не делающая функция в процессе разработки программы бывает полезна в качестве "хранителя места". Если тип результата опущен, то предполагается, что функция возвращает значение типа `int`.

Любая программа - это просто совокупность определений переменных и функций. Связи между функциями осуществляются через аргументы, возвращаемые значения и внешние переменные. В исходном файле функции могут располагаться в любом порядке; исходную программу можно разбивать на любое число файлов, но так, чтобы ни одна из функций не оказалась разрезанной.

Инструкция *return* реализует механизм возврата результата от вызываемой функции к вызывающей. За словом `return` может следовать любое выражение:

return *выражение*;

Если потребуется, *выражение* будет приведено к возвращаемому типу функции. Часто *выражение* заключают в скобки, но они не обязательны.

Вызывающая функция вправе проигнорировать возвращаемое значение. Более того, *выражение* в return может отсутствовать, и тогда вообще никакое значение не будет возвращено в вызывающую функцию. Управление возвращается в вызывающую функцию без результирующего значения также и в том случае, когда вычисления достигли "конца" (т. е. последней закрывающей фигурной скобки функции). Не запрещена (но должна вызывать настороженность) ситуация, когда в одной и той же функции одни return имеют при себе выражения, а другие - не имеют. Во всех случаях, когда функция "забыла" передать результат в return, она обязательно выдаст "мусор".

Функция main в программе поиска по образцу возвращает в качестве результата количество найденных строк. Это число доступно той среде, из которой данная программа была вызвана.

Механизмы компиляции и загрузки Си-программ, расположенных в нескольких исходных файлах, в разных системах могут различаться. В системе UNIX, например, эти работы выполняет упомянутая в главе 1 команда `cc`. Предположим, что три функции нашего последнего примера расположены в трех разных файлах: `main.c`, `getline.c` и `strindex.c`. Тогда команда

```
cc main.c getline.c strindex.c
```

скомпилирует указанные файлы, поместив результат компиляции в файлы объектных модулей `main.o`, `getline.o` и `strindex.o`, и затем загрузит их в исполняемый файл `a.out`. Если обнаружилась ошибка, например в файле `main.c`, то его можно скомпилировать снова и результат загрузить ранее полученными объектными файлами, выполнив следующую команду:

```
cc main.c getline.o strindex.o
```

Команда `cc` использует стандартные расширения файлов ".c" и ".o", чтобы отличать исходные файлы от объектных.

Упражнение 4.1. Напишите функцию strindex(s,t), которая выдает позицию *самого правого* вхождения t в s или -1, если вхождения не обнаружено.

4.2. Функции, возвращающие нецелые значения

В предыдущих примерах функции либо вообще не возвращали результирующих значений (void), либо возвращали значения типа int. А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sin и cos, возвращают значения типа double; другие специальные функции могут выдавать значения еще каких-то типов. Чтобы проиллюстрировать, каким образом функция может возвратить нецелое значение, напишем функцию atof(s), которая переводит строку s в соответствующее число с плавающей точкой двойной точности. Функция atof представляет собой расширение функции atoi, две версии которой были рассмотрены в главах 2 и 3. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Наша версия *не* является высококачественной программой преобразования вводимых чисел; такая программа потребовала бы заметно больше памяти. Функция atof входит в стандартную библиотеку программ; ее описание содержится в заголовочном файле <stdlib.h>.

Прежде всего отметим, что объявлять тип возвращаемого значения должна сама atof, так как этот тип не есть int. Указатель типа задается перед именем функции.

```
#include <ctype.h>

/* atof: преобразование строки s в double */
double atof (char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace (s[i]); i++)
        ;       /* игнорирование левых символов-разделителей */
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit (s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]; i++) {
        val = 10.0 * val + (s[i] - '0');
```

```
        power *= 10.0;
    }
    return sign * val / power;
}
```

Кроме того, важно, чтобы вызывающая программа знала, что atof возвращает нецелое значение. Один из способов обеспечить это - явно описать atof в вызывающей программе. Подобное описание демонстрируется ниже в программе простенького калькулятора (достаточного для проверки баланса чековой книжки), который каждую вводимую строку воспринимает как число, прибавляет его к текущей сумме и печатает ее новое значение.

```
#include <stdio.h>
#define MAXLINE 100

/* примитивный калькулятор */
main()
{
    double sum, atof (char[]);
    char line[MAXLINE];
    int getline (char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf ("\t%g\n", sum += atof(line));
    return 0;
}
```

В объявлении

```
double sum, atof (char[]);
```

говорится, что sum - переменная типа double, a atof - функция, которая принимает один аргумент типа char[] и возвращает результат типа double.

Объявление и определение функции atof должны соответствовать друг другу. Если в одном исходном файле сама функция atof и обращение к ней в main имеют разные типы, то это несоответствие будет зафиксировано компилятором как ошибка. Но если функция atof была скомпилирована отдельно (что более вероятно), то несоответствие типов не будет обнаружено, и atof возвратит значение типа double, которое функция main воспримет как int, что приведет к бессмысленному результату.

Это последнее утверждение, вероятно, вызовет у вас удивление, поскольку ранее говорилось о необходимости соответствия объявлений

и определений. Причина несоответствия, возможно, будет следствием того, что вообще отсутствует прототип функции, и функция неявно объявляется при первом своем появлении в выражении, как, например, в

```
sum += atof(line)
```

Если в выражении встретилось имя, нигде ранее не объявленное, за которым следует открывающая скобка, то такое имя по контексту считается именем функции, возвращающей результат типа int; при этом относительно ее аргументов ничего не предполагается. Если в объявлении функции аргументы не указаны, как в

```
double atof();
```

то и в этом случае считается, что ничего об аргументах atof не известно, и все проверки на соответствие ее параметров будут выключены. Предполагается, что такая специальная интерпретация пустого списка позволит новым компиляторам транслировать старые Си-программы. Но в новых программах пользоваться этим - не очень хорошая идея. Если у функции есть аргументы, опишите их, если их нет, используйте слово void.

Располагая соответствующим образом описанной функцией atof, мы можем написать функцию atoi, преобразующую строку символов в целое значение, следующим образом:

```
/* atoi: преобразование строки s в int с помощью atof */
int atoi (char s[])
{
  double atof (char s[]);

  return (int) atof (s);
}
```

Обратите внимание на вид объявления и инструкции return. Значение выражения в

return *выражение;*

перед тем, как оно будет возвращено в качестве результата, приводится к типу функции. Следовательно, поскольку функция atoi возвращает значение int, результат вычисления atof типа double в инструкции return автоматически преобразуется в тип int. При преобразовании возможна потеря информации, и некоторые компиляторы предупреждают об этом. Оператор приведения явно указывает на необходимость преобразования типа и подавляет любое предупреждающее сообщение.

Упражнение 4.2. Дополните функцию atof таким образом, чтобы она справлялась с числами вида

```
123.45e-6
```

в которых после мантиссы может стоять е (или Е) с последующим порядком (быть может, со знаком).

4.3. Внешние переменные

Программа на Си обычно оперирует с множеством внешних объектов: переменных и функций. Прилагательное "внешний" (external) противоположно прилагательному "внутренний", которое относится к аргументам и переменным, определяемым внутри функций. Внешние переменные определяются вне функций и потенциально доступны для многих функций. Сами функции всегда являются внешними объектами, поскольку в Си запрещено определять функции внутри других функций. По умолчанию одинаковые внешние имена, используемые в разных файлах, относятся к одному и тому же внешнему объекту (функции). (В стандарте это называется *редактированием внешних связей (external linkage*[1]).) В этом смысле внешние переменные похожи на области COMMON в фортране и на переменные самого внешнего блока в Паскале. Позже мы покажем, как внешние функции и переменные сделать видимыми только внутри одного исходного файла.

Поскольку внешние переменные доступны всюду, их можно использовать в качестве связующих данных между функциями как альтернативу связей через аргументы и возвращаемые значения. Для любой функции внешняя переменная доступна по ее имени, если это имя было должным образом объявлено.

Если число переменных, совместно используемых функциями, велико, связи между последними через внешние переменные могут оказаться более удобными и эффективными, чем длинные списки аргументов. Но, как отмечалось в главе 1, к этому заявлению следует относиться критически, поскольку такая практика ухудшает структуру программы и приводит к слишком большому числу связей между функциями по данным.

Внешние переменные полезны, так как они имеют большую область действия и время жизни. Автоматические переменные существуют только внутри функции, они возникают в момент входа в функцию и исчезают при выходе из нее. Внешние переменные, напротив, существуют постоянно, так что их значения сохраняются и между обращениями к функциям. Таким образом, если двум функциям приходится пользоваться одними и теми же данными и ни одна из них не вызывает другую, то часто

---

[1] Сейчас ужо и русский язык прочно вошло слово "*линкование*". — *Примеч. ред.*

бывает удобно оформить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно через аргументы.

В связи с приведенными рассуждениями разберем пример. Поставим себе задачу написать программу-калькулятор, понимающую операторы +, -, * и /. Такой калькулятор легче будет написать, если ориентироваться на польскую, а не инфиксную запись выражений. (Обратная польская запись применяется в некоторых карманных калькуляторах и в таких языках, как Forth и Postscript.)

В обратной польской записи каждый оператор следует за своими операндами. Выражение в инфиксной записи, скажем

```
(1 - 2) * (4 + 5)
```

в польской записи представляется как

```
1 2 - 4 5 + *
```

Скобки не нужны, неоднозначности в вычислениях не бывает, поскольку известно, сколько операндов требуется для каждого оператора.

Реализовать нашу программу весьма просто. Каждый операнд посылается в стек; если встречается оператор, то из стека берется соответствующее число операндов (в случае бинарных операторов два) и выполняется операция, после чего результат посылается в стек. В нашем примере числа 1 и 2 посылаются в стек, затем замещаются на их разность -1. Далее в стек посылаются числа 4 и 5, которые затем заменяются их суммой (9). Числа -1 и 9 заменяются в стеке их произведением (т. е. -9). Встретив символ новой строки, программа извлекает значение из стека и печатает его.

Таким образом, программа состоит из цикла, обрабатывающего на каждом своем шаге очередной встречаемый оператор или операнд:

```
while (следующий элемент не конец-файла)
  if (число)
    послать его в стек
  else if (оператор)
    взять из стека операнды
    выполнить операцию
    результат послать в стек
  else if (новая-строка)
    взять с вершины стека число и напечатать
  else
    ошибка
```

Операции "послать в стек" и "взять из стека" сами по себе тривиальны, однако по мере добавления к ним механизмов обнаружения и нейтрализации ошибок становятся достаточно длинными. Поэтому их лучше

оформить в виде отдельных функций, чем повторять соответствующий код по всей программе. И конечно необходимо иметь отдельную функцию для получения очередного оператора или операнда.

Главный вопрос, который мы еще не рассмотрели, - это вопрос о том, где расположить стек и каким функциям разрешить к нему прямой доступ. Стек можно расположить в функции main и передавать сам стек и текущую позицию в нем в качестве аргументов функциям push ("послать в стек") и pop ("взять из стека"). Но функции main нет дела до переменных, относящихся к стеку, - ей нужны только операции по помещению чисел в стек и извлечению их оттуда. Поэтому мы решили стек и связанную с ним информацию хранить во внешних переменных, доступных для функций push и pop, но не доступных для main.

Переход от эскиза к программе достаточно легок. Если теперь программу представить как текст, расположенный в одном исходном файле, она будет иметь следующий вид:

```
#include /* могут быть в любом количестве */
#define  /* могут быть в любом количестве */

объявления функций для main

main () {...}

внешние переменные для push и pop

void push (double f) {...}
double pop (void) {...}

int getop(char s[]) {...}

подпрограммы, вызываемые функцией getop
```

Позже мы обсудим, как текст этой программы можно разбить на два или большее число файлов.

Функция main - это цикл, содержащий большой переключатель switch, передающий управление на ту или иную ветвь в зависимости от типа оператора или операнда. Здесь представлен более типичный случай применения переключателя switch по сравнению с рассмотренным в параграфе 3.4.

```
#include <stdio.h>
#include <stdlib.h>  /* для atof() */

#define MAXOP 100    /* макс. размер операнда или оператора */
#define NUMBER '0'   /* признак числа */
```

```
int getop (char []);
void push (double);
double pop (void);

/* калькулятор с обратной польской записью */
main ()
{
   int type;
   double op2;
   char s[MAXOP];

   while ((type = getop (s)) != EOF) {
      switch (type) {
      case NUMBER:
         push (atof (s));
         break;
      case '+':
         push (pop() + pop());
         break;
      case '*':
         push (pop () * pop ());
         break;
      case '-':
         op2 = pop ();
         push (pop () - op2);
         break;
      case '/':
         op2 = pop ();
         if (op2 != 0.0)
            push (pop () / op2);
         else
            printf("ошибка: деление на нуль\n");
         break;
      case '\n' :
         printf("\t%.8g\n", pop ());
         break;
      default:
         printf("ошибка: неизвестная операция %s\n", s);
         break;
      }
   }
   return 0;
}
```

Так как операторы + и * коммутативны, порядок, в котором операнды берутся из стека, не важен, однако в случае операторов - и /, левый и правый операнды должны различаться. Так, в

```
push(pop() - pop()); /* НЕПРАВИЛЬНО */
```

очередность обращения к pop не определена. Чтобы гарантировать правильную очередность, необходимо первое значение из стека присвоить временной переменной, как это и сделано в main.

```
#define MAXVAL 100     /* максимальная глубина стека */

int sp = 0;            /* следующая свободная позиция в стеке */
double val[MAXVAL];    /* стек */

/* push: положить значение f в стек */
void push(double f)
{
  if (sp < MAXVAL)
    val[sp++] = f;
  else
    printf( "ошибка: стек полон, %g не помещается\n", f);
}

/* pop: взять с вершины стека и выдать в качестве результата */
double pop(void)
{
  if (sp > 0)
    return val[--sp];
  else {
    printf("ошибка: стек пуст\n");
    return 0.0;
  }
}
```

Переменная считается внешней, если она определена вне функции. Таким образом, стек и индекс стека, которые должны быть доступны и для push, и для pop, определяются вне этих функций. Но main не использует ни стек, ни позицию в стеке, и поэтому их представление может быть скрыто от main.

Займемся реализацией getop - функции, получающей следующий оператор или операнд. Нам предстоит решить довольно простую задачу. Более точно: требуется пропустить пробелы и табуляции; если следующий символ - не цифра и не десятичная точка, то нужно выдать его; в противном

случае надо накопить строку цифр с десятичной точкой, если она есть, и выдать число NUMBER в качестве результата.

```
#include<ctype.h>

int getch(void);
void ungetch(int);

/* getop: получает следующий оператор или операнд */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;    /* не число */
    i = 0;
    if (isdigit(c))  /* накапливаем целую часть */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')    /* накапливаем дробную часть */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Как работают функции getch и ungetch? Во многих случаях программа не может "сообразить", прочла ли она все, что требуется, пока не прочтет лишнего. Так, накопление числа производится до тех пор, пока не встретится символ, отличный от цифры. Но это означает, что программа прочла на один символ больше, чем нужно, и последний символ нельзя включать в число.

Эту проблему можно было бы решить при наличии обратной чтению операции “положить-назад”, с помощью которой можно было бы вернуть ненужный символ. Тогда каждый раз, когда программа считает на один символ больше, чем требуется, эта операция возвращала бы его вводу, и остальная часть программы могла бы вести себя так, будто этот символ

вовсе и не читался. К счастью, описанный механизм обратной посылки символа легко моделируется с помощью пары согласованных друг с другом функций, из которых getch поставляет очередной символ из ввода, а ungetch отправляет символ назад во входной поток, так что при следующем обращении к getch мы вновь его получим.

Нетрудно догадаться, как они работают вместе. Функция ungetch запоминает посылаемый назад символ в некотором буфере, представляющем собой массив символов, доступный для обеих этих функций; getch читает из буфера, если там что-то есть, или обращается к getchar, если буфер пустой. Следует предусмотреть индекс, указывающий на положение текущего символа в буфере.

Так как функции getch и ungetch совместно используют буфер и индекс, значения последних должны между вызовами сохраняться. Поэтому буфер и индекс должны быть внешними по отношению к этим программам, и мы можем записать getch, ungetch и общие для них переменные в следующем виде:

```
#define BUFSIZE 100

char buf[BUFSIZE];      /* буфер для ungetch */
int bufp = 0;           /* след. свободная позиция в буфере */

int getch(void)         /* взять (возможно возвращенный) символ */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)     /* вернуть символ на ввод */
{
    if (bufp >= BUFSIZE)
        printf ("ungetch: слишком много символов\n");
    else
        buf[bufp++] = c;
}
```

Стандартная библиотека включает функцию ungetc, обеспечивающую возврат одного символа (см. главу 7). Мы же, чтобы проиллюстрировать более общий подход, для запоминания возвращаемых символов использовали массив.

Упражнение 4.3. Исходя из предложенной нами схемы, дополните программу-калькулятор таким образом, чтобы она "понимала" оператор получения остатка от деления (%) и отрицательные числа.

Упражнение 4.4. Добавьте команды, с помощью которых можно было бы печатать верхний элемент стека (с сохранением его в стеке), дублировать его в стеке, менять местами два верхних элемента стека. Введите команду очистки стека.

Упражнение 4.5. Предусмотрите возможность использования в программе библиотечных функций sin, exp и pow. См. библиотеку <math.h> в приложении B (параграф 4).

Упражнение 4.6. Введите команды для работы с переменными (легко обеспечить до 26 переменных, каждая из которых имеет имя, представленное одной буквой латинского алфавита). Добавьте переменную, предназначенную для хранения самого последнего из напечатанных значений.

Упражнение 4.7. Напишите программу ungets(s), возвращающую строку s во входной поток. Должна ли ungets "знать" что-либо о переменных buf и bufp, или ей достаточно пользоваться только функцией ungetch?

Упражнение 4.8. Предположим, что число символов, возвращаемых назад, не превышает 1. Модифицируйте с учетом этого факта функции getch и ungetch.

Упражнение 4.9. В наших функциях не предусмотрена возможность возврата EOF. Подумайте, что надо сделать, чтобы можно было возвращать EOF, и скорректируйте соответственно программу.

Упражнение 4.10. В основу программы калькулятора можно положить применение функции getline, которая читает целиком строку; при этом отпадает необходимость в getch и ungetch. Напишите программу, реализующую этот подход.

4.4. Области видимости

Функции и внешние переменные, из которых состоит Си-программа, каждый раз компилировать все вместе нет никакой необходимости. Исходный текст можно хранить в нескольких файлах. Ранее скомпилированные программы можно загружать из библиотек. В связи с этим возникают следующие вопросы:

- Как писать объявления, чтобы на протяжении компиляции используемые переменные были должным образом объявлены?

- В каком порядке располагать объявления, чтобы во время загрузки все части программы оказались связаны нужным образом?
- Как организовать объявления, чтобы они имели лишь одну копию?
- Как инициализировать внешние переменные?

Начнем с того, что разобьем программу-калькулятор на несколько файлов. Конечно, эта программа слишком мала, чтобы ее стоило разбивать на файлы, однако разбиение нашей программы позволит продемонстрировать проблемы, возникающие в больших программах.

Областью видимости имени считается часть программы, в которой это имя можно использовать. Для автоматических переменных, объявленных в начале функции, областью видимости является функция, в которой они объявлены. Локальные переменные разных функций, имеющие, однако, одинаковые имена, никак не связаны друг с другом. То же утверждение справедливо и в отношении параметров функции, которые фактически являются локальными переменными.

Область действия внешней переменной или функции простирается от точки программы, где она объявлена, до конца файла, подлежащего компиляции. Например, если main, sp, val, push и pop определены в одном файле в указанном порядке, т. е.

```
main() {...}

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

то к переменным sp и val можно адресоваться из push и pop просто по их именам; никаких дополнительных объявлений для этого не требуется. Заметим, что в main эти имена не видимы так же, как и сами push и pop.

Однако, если на внешнюю переменную нужно сослаться до того, как она определена, или если она определена в другом файле, то ее объявление должно быть помечено словом extern.

Важно отличать *объявление* внешней переменной от ее *определения*. Объявление объявляет свойства переменной (прежде всего ее тип), а определение, кроме того, приводит к выделению для нее памяти. Если строки

```
int sp;
double val[MAXVAL];
```

расположены вне всех функций, то они *определяют* внешние переменные

sp и val, т. е. отводят для них память, и, кроме того, служат объявлениями для остальной части исходного файла. А вот строки

```
extern int sp;
extern double val[];
```

объявляют для оставшейся части файла, что sp - переменная типа int, a val - массив типа double (размер которого определен где-то в другом месте); при этом ни переменная, ни массив не создаются, и память им не отводится.

На всю совокупность файлов, из которых состоит исходная программа, для каждой внешней переменной должно быть одно-единственное *определение;* другие файлы, чтобы получить доступ к внешней переменной, должны иметь в себе объявление extern. (Впрочем, объявление extern можно поместить и в файл, в котором содержится определение.) В определениях массивов необходимо указывать их размеры, что в объявлениях extern не обязательно.

Инициализировать внешнюю переменную можно только в определении.

Хотя вряд ли стоит организовывать нашу программу таким образом, но мы определим push и pop в одном файле, a val и sp - в другом, где их и инициализируем. При этом для установления связей понадобятся такие определения и объявления:

В файле1:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

В файле2:

```
int sp = 0;
double val[MAXVAL];
```

Поскольку объявления extern находятся в начале *файла1* и вне определений функций, их действие распространяется на все функции, причем одного набора объявлений достаточно для всего *файла1*. Та же организация extern-объявлений необходима и в случае, когда программа состоит из одного файла, но определения sp и val расположены после их использования.

4.5. Заголовочные файлы

Теперь представим себе, что компоненты программы-калькулятора имеют существенно большие размеры, и зададимся вопросом, как в этом случае распределить их по нескольким файлам. Программу main помес-

тим в файл, который мы назовем main.c; push, pop и их переменные расположим во втором файле, stack.c; a getop - в третьем, getop.c. Наконец, getch и ungetch разместим в четвертом файле getch.c; мы отделили их от остальных функций, поскольку в реальной программе они будут получены из заранее скомпилированной библиотеки.

Существует еще один момент, о котором следует предупредить читателя, - определения и объявления совместно используются несколькими файлами. Мы бы хотели, насколько это возможно, централизовать эти объявления и определения так, чтобы для них существовала только одна копия. Тогда программу в процессе ее развития будет легче и исправлять, и поддерживать в нужном состоянии. Для этого общую информацию расположим в *заголовочном файле* calc.h, который будем по мере необходимости включать в другие файлы. (Строка #include описывается в параграфе 4.1 1.) В результате получим программу, файловая структура которой показана ниже:

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char[]);
int getch(void);
void ungetch(int);
```

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
...
}
```

getop.c:

```
#include <stdio.h>
#include <stype.h>
«include "calc.h"
getop (){
...
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
...
}
double pop(void) {
...
}
```

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf [BUFSIZE];
int bufp = 0;
int getch(void) {
...
}
void ungetch(int) {
...
}
```

Неизбежен компромисс между стремлением, чтобы каждый файл владел только той информацией, которая ему необходима для работы, и тем, что на практике иметь дело с большим количеством заголовочных файлов довольно трудно. Для программ, не превышающих некоторого среднего размера, вероятно, лучше всего иметь один заголовочный файл, в котором собраны вместе все объекты, каждый из которых используется в двух различных файлах; так мы здесь и поступили. Для программ больших размеров потребуется более сложная организация с большим числом заголовочных файлов.

4.6. Статические переменные

Переменные sp и val в файле stack.c, а также buf и bufp в getch.c находятся в личном пользовании функций этих файлов, и нет смысла открывать к ним доступ кому-либо еще. Указание static, примененное к внешней переменной или функции, ограничивает область видимости соответствующего объекта концом файла. Это способ скрыть имена. Так, переменные buf и bufp должны быть внешними, поскольку их совместно используют функции getch и ungetch, но их следует сделать невидимыми для "пользователей" функций getch и ungetch.

Статическая память специфицируется словом static, которое помещается перед обычным объявлением. Если рассматриваемые нами две функции и две переменные компилируются в одном файле, как в показанном ниже примере:

```
static char buf[BUFSIZE];   /* буфер для ungetch */
static int bufp = 0;        /* след. свободная позиция в buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

то никакая другая программа не будет иметь доступ ни к buf, ни к bufp, и этими именами можно свободно пользоваться в других файлах для совсем иных целей. Точно так же, помещая указание static перед объявлениями переменных sp и val, с которыми работают только push и pop, мы можем скрыть их от остальных функций.

Указание static чаще всего используется для переменных, но с равным успехом его можно применять и к функциям. Обычно имена функций глобальны и видимы из любого места программы. Если же функция помечена словом static, то ее имя становится невидимым вне файла, в котором она определена.

Объявление static можно использовать и для внутренних переменных. Как и автоматические переменные, внутренние статические переменные локальны в функциях, но в отличие от автоматических они не возникают только на период работы функции, а существуют постоянно. Это значит, что внутренние статические переменные обеспечивают постоянное сохранение данных внутри функции.

Упражнение 4.11. Модифицируйте функцию getop так, чтобы отпала необходимость в функции ungetch. Подсказка: используйте внутреннюю статическую переменную.

4.7. Регистровые переменные

Объявление register сообщает компилятору, что данная переменная будет интенсивно использоваться. Идея состоит в том, чтобы переменные, объявленные register, разместить на регистрах машины, благодаря чему программа, возможно, станет более короткой и быстрой. Однако компилятор имеет право проигнорировать это указание.

Объявление register выглядит следующим образом:

```
register int x;
register char c;
```

и т. д. Объявление register может применяться только к автоматическим переменным и к формальным параметрам функции. Для последних это выглядит так:

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

На практике существуют ограничения на регистровые переменные, что связано с возможностями аппаратуры. Располагаться в регистрах может лишь небольшое число переменных каждой функции, причем только определенных типов. Избыточные объявления register ни на что не влияют, так как игнорируются в отношении переменных, которым не хватило регистров или которые нельзя разместить на регистре. Кроме того, применительно к регистровой переменной независимо от того, выделен на самом деле для нее регистр или нет, не определено понятие адреса (см. главу 5). Конкретные ограничения на количество и типы регистровых переменных зависят от машины.

4.8. Блочная структура

Поскольку функции в Си нельзя определять внутри других функций, он не является языком, допускающим блочную структуру программы в том смысле, как это допускается в Паскале и подобных ему языках. Но переменные внутри функций можно определять в блочно-структурной манере. Объявления переменных (вместе с инициализацией) разрешено помещать не только в начале функции, но и после *любой* левой фигурной скобки, открывающей составную инструкцию. Переменная, описанная таким способом, "затеняет" переменные с тем же именем, расположенные в объемлющих блоках, и существует вплоть до соответствующей правой фигурной скобки. Например, в

```
if (n > 0) {
    int i; /* описание новой переменной i */

    for (i = 0; i < n; i++)
        ...
}
```

областью видимости переменной i является ветвь `if`, выполняемая при n>0; и эта переменная никакого отношения к любым i, расположенным вне данного блока, не имеет. Автоматические переменные, объявленные и инициализируемые в блоке, инициализируются каждый раз при входе в блок. Переменные `static` инициализируются только один раз при первом входе в блок.

Автоматические переменные и формальные параметры также "затеняют" внешние переменные и функции с теми же именами. Например, в

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

x внутри функции f рассматривается как параметр типа `double`, в то время как вне f это внешняя переменная типа `int`. То же самое можно сказать и о переменной y.

С точки зрения стиля программирования, лучше не пользоваться одними и теми же именами для разных переменных, поскольку слишком велика возможность путаницы и появления ошибок.

4.9. Инициализация

Мы уже много раз упоминали об инициализации, но всегда лишь по случаю, в ходе обсуждения других вопросов. В этом параграфе мы суммируем все правила, определяющие инициализацию памяти различных классов.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление; автоматические и регистровые переменные имеют неопределенные начальные значения ("мусор").

Скалярные переменные можно инициализировать в их определениях, помещая после имени знак = и соответствующее выражение:

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L; /* день в миллисекундах */
```

Для внешних и статических переменных инициализирующие выражения должны быть константными, при этом инициализация осуществляется только один раз до начала выполнения программы. Инициализация автоматических и регистровых переменных выполняется каждый раз при входе в функцию или блок. Для таких переменных инициализирующее выражение - не обязательно константное. Это может быть любое выражение, использующее ранее определенные значения, включая даже и вызовы функций. Например, в программе бинарного поиска, описанной в параграфе 3.3, инициализацию можно записать так:

```
int binsearch(int x, int v[], int n)
{
  int low = 0;
  int high = n - 1;
  int mid;
    ...
}
```

а не так:

```
int low, high, mid;

low = 0;
high = n - 1;
```

В сущности, инициализация автоматической переменной - это более короткая запись инструкции присваивания. Какая запись предпочтительнее - в большой степени дело вкуса. До сих пор мы пользовались главным образом явными присваиваниями, поскольку инициализация в объявлениях менее заметна и дальше отстоит от места использования переменной.

Массив можно инициализировать в его определении с помощью заключенного в фигурные скобки списка инициализаторов, разделенных запятыми. Например, чтобы инициализировать массив days, элементы которого суть количества дней в каждом месяце, можно написать:

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Если размер массива не указан, то длину массива компилятор вычисляет по числу заданных инициализаторов; в нашем случае их количество равно 12.

Если количество инициализаторов меньше числа, указанного в определении длины массива, то для внешних, статических и автоматических переменных оставшиеся элементы будут нулевыми. Задание слишком большого числа инициализаторов считается ошибкой. В языке нет возможности ни задавать повторения инициализатора, ни инициализовать средние элементы массива без задания всех предшествующих значений.

Инициализация символьных массивов - особый случай: вместо конструкции с фигурными скобками и запятыми можно использовать строку символов. Например, возможна такая запись:

```
char pattern[] = "ould";
```

представляющая собой более короткий эквивалент записи

```
char pattern[] = {'o', V, 'l', 'd', '\0'};
```

В данном случае размер массива равен пяти (четыре обычных символа и завершающий символ '\0').

4.10. Рекурсия

В Си допускается рекурсивное обращение к функциям, т. е. функция может обращаться сама к себе, прямо или косвенно. Рассмотрим печать числа в виде строки символов. Как мы упоминали ранее, цифры генерируются в обратном порядке - младшие цифры получаются раньше старших, а печататься они должны в правильной последовательности.

Проблему можно решить двумя способами. Первый - запомнить цифры в некотором массиве в том порядке, как они получались, а затем напечатать их в обратном порядке; так это и было сделано в функции itoa, рассмотренной в параграфе 3.6. Второй способ - воспользоваться рекурсией, при которой printd сначала вызывает себя, чтобы напечатать все старшие цифры, и затем печатает последнюю младшую цифру. Эта программа, как и предыдущий ее вариант, при использовании самого большого по модулю отрицательного числа работает неправильно.

```
#include <stdio.h>

/* printd: печатает n как целое десятичное число */
void printd(int n)
{
  if (n < 0) {
    putchar('-');
    n = -n;
  }
  if (n / 10)
    printd(n / 10);
  putchar(n % 10 + '0' );
}
```

Когда функция рекурсивно обращается сама к себе, каждое следующее обращение сопровождается получением ею нового полного набора автоматических переменных, независимых от предыдущих наборов. Так, в обращении printd(123) при первом вызове аргумент n = 123, при втором – printd получает аргумент 12, при третьем вызове - значение 1. Функция printd на третьем уровне вызова печатает 1 и возвращается на второй уровень, после чего печатает цифру 2 и возвращается на первый уровень. Здесь она печатает 3 и заканчивает работу.

Следующий хороший пример рекурсии - это быстрая сортировка, предложенная Ч. А. Р. Хоаром в 1962 г. Для заданного массива выбирается один элемент, который разбивает остальные элементы на два подмножества - те, что меньше, и те, что не меньше него. Та же процедура рекурсивно применяется и к двум полученным подмножествам. Если в подмножестве менее двух элементов, то сортировать нечего, и рекурсия завершается.

Наша версия быстрой сортировки, разумеется, не самая быстрая среди всех возможных, но зато одна из самых простых. В качестве делящего элемента мы используем серединный элемент.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(int v[], int left, int right)
{
  int i, last;
  void swap(int v[], int i, int j);

  if (left >= right)       /* ничего не делается, если */
    return;                /* в массиве менее двух элементов */
  swap(v, left, (left + right)/2);    /* делящий элемент */
  last = left;                        /* переносится в v[0] */
  for(i = left+1; i <= right; i++)   /* деление на части */
```

```
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);    /* перезапоминаем делящий элемент */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

В нашей программе операция перестановки оформлена в виде отдельной функции (swap), поскольку встречается в qsort трижды.

```
/* swap: поменять местами v[i] и v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Стандартная библиотека имеет функцию qsort, позволяющую сортировать объекты любого типа.

Рекурсивная программа не обеспечивает ни экономии памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, а часто намного легче для написания и понимания. Такого рода программы особенно удобны для обработки рекурсивно определяемых структур данных вроде деревьев; с хорошим примером на эту тему вы познакомитесь в параграфе 6.5.

Упражнение 4.12. Примените идеи, которые мы использовали в printd, для написания рекурсивной версии функции itoa; иначе говоря, преобразуйте целое число в строку цифр с помощью рекурсивной программы.

Упражнение 4.13. Напишите рекурсивную версию функции reverse(s), переставляющую элементы строки в ту же строку в обратном порядке.

4.11. Препроцессор языка Си

Некоторые возможности языка Си обеспечиваются препроцессором, который работает на первом шаге компиляции. Наиболее часто используются две возможности: #include, вставляющая содержимое некоторого файла во время компиляции, #define, заменяющая одни текстовые по-

следовательности на другие. В этом параграфе обсуждаются условная компиляция и макроподстановка с аргументами.

4.11.1. Включение файла

Средство `#include` позволяет, в частности, легко манипулировать наборами «define и объявлений. Любая строка вида

`#include` *"имя-файла"*

или

`#include` *<имя-файла>*

заменяется содержимым файла с именем *имя-файла.* Если *имя-файла* заключено в двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось или *имя-файла* заключено в угловые скобки < и >, то поиск осуществляется по определенным в реализации правилам. Включаемый файл сам может содержать в себе строки `#include`.

Часто исходные файлы начинаются с нескольких строк «include, ссылающихся на общие инструкции `#define` и объявления `extern` или прототипы нужных библиотечных функций из заголовочных файлов вроде `<stdio.h>`. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к заголовкам, зависят от конкретной реализации.)

Средство `#include` - хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

4.11.2. Макроподстановка

Определение макроподстановки имеет вид:

`#define` *имя замещающий-текст*

Макроподстановка используется для простейшей замены: во всех местах, где встречается лексема *имя,* вместо нее будет помещен *замещающий-текст.* Имена в `#define` задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст завершает строку, в которой расположено слово `#define`, но в длинных определениях его можно продолжить на следующих строках, поставив в конце каждой продолжаемой строки обратную наклонную черту \. Область видимости имени, определенного в `#define`,

простирается от данного определения до конца файла. В определении макроподстановки могут фигурировать более ранние #define-определения. Подстановка осуществляется только для тех имен, которые расположены вне текстов, заключенных в кавычки. Например, если YES определено с помощью #define, то никакой подстановки в printf("YES") или в YESMAN выполнено не будет.

Любое имя можно определить с произвольным замещающим текстом. Например,

```
tfdefine forever for(;;)            /* бесконечный цикл */
```

определяет новое слово forever для бесконечного цикла.

Макроподстановку можно определить с аргументами, вследствие чего замещающий текст будет варьироваться в зависимости от задаваемых параметров. Например, определим max следующим образом:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Хотя обращения к max выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае A и B) будет заменяться соответствующим ему аргументом. Так, строка

```
x = max(p+q, r+s);
```

будет заменена на строку

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Поскольку аргументы допускают любой вид замены, указанное определение max подходит для данных любого типа, так что не нужно писать разные max для данных разных типов, как это было бы в случае задания с помощью функций.

Если вы внимательно проанализируете работу max, то обнаружите некоторые подводные камни. Выражения вычисляются дважды, и если они вызывают побочный эффект (из-за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max(i++, j++)                       /* НЕВЕРНО */
```

вызовет увеличение i и j дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Задумайтесь, что случится, если при определении

```
#define square(x) x*x               /* НЕВЕРНО */
```

вызвать square(z+1).

Тем не менее макросредства имеют свои достоинства. Практическим примером их использования является частое применение getchar и putchar из <stdio.h>, реализованных с помощью макросов, чтобы избежать расходов времени от вызова функции на каждый обрабатываемый символ. Функции в <ctype.h> обычно также реализуются с помощью макросов.

Действие ttdefine можно отменить с помощью flundef:

```
#undef getchar

int getchar(void) { ... }
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем.

Имена формальных параметров не заменяются, если встречаются в заключенных в кавычки строках. Однако, если в замещающем тексте перед формальным параметром стоит знак #, этот параметр будет заменен на аргумент, заключенный в кавычки. Это может сочетаться с конкатенацией (склеиванием) строк, например, чтобы создать макрос отладочного вывода:

```
#undef dprint(expr) printf(#expr " = %g\n", expr)
```

Обращение к

```
dprint(x/y);
```

развернется в

```
printf("x/y" " = %g\n", x/y);
```

а в результате конкатенации двух соседних строк получим

```
printf("x/y = %g\n", x/y);
```

Внутри фактического аргумента каждый знак " заменяется на \", а каждая \ на \\, так что результат подстановки приводит к правильной символьной константе.

Оператор ## позволяет в макрорасширениях конкатенировать аргументы. Если в замещающем тексте параметр соседствует с ##, то он заменяется соответствующим ему аргументом, а оператор ## и окружающие его символы-разделители выбрасываются. Например, в макроопределении paste конкатенируются два аргумента

```
#define paste(front, back) front Sit back
```

так что paste(name, 1) сгенерирует имя name1.

Правила вложенных использований оператора ## не определены; другие подробности, относящиеся к ##, можно найти в приложении А.

Упражнение 4.14. Определите swap(t,x,y) в виде макроса, который осуществляет обмен значениями указанного типа t между аргументами x и y. (Примените блочную структуру.)

4.11.3. Условная компиляция

Самим ходом препроцессирования можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции.

Вычисляется константное целое выражение, заданное в строке #if. Это выражение не должно содержать ни одного оператора sizeof или приведения к типу и ни одной enum-константы. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до #endif, или #elif, или #else. (Инструкция препроцессора «elif похожа на else if.) Выражение defined(*имя*) в «if есть 1, если *имя* было определено, и О в противном случае.

Например, чтобы застраховаться от повторного включения заголовочного файла hdr.h, его можно оформить следующим образом:

```
«if !defined(HDR)
#defineHDR

/* здесь содержимое hdr.h */

«endif
```

При первом включении файла hdr.h будет определено имя HDR, а при последующих включениях препроцессор обнаружит, что имя HDR уже определено, и перескочит сразу на #endif. Этот прием может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя.

Вот пример цепочки проверок имени SYSTEM, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   «define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
«else
```

```
    #defineHDR"default.h"
#endif
#includeHDR
```

Инструкции flifdefи #ifndefспециально предназначены для проверки того, определено или нет заданное в них имя. И следовательно, первый пример, приведенный выше для иллюстрации #if, можно записать и в таком виде:

```
tfifndef    HDR
#defineHDR

/* здесь содержимое hdr.h */

tfendif
```

Глава 5

Указатели и массивы

Указатель - это переменная, содержащая адрес переменной. Указатели широко применяются в Си - отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом; в данной главе мы рассмотрим эту зависимость и покажем, как ею пользоваться.

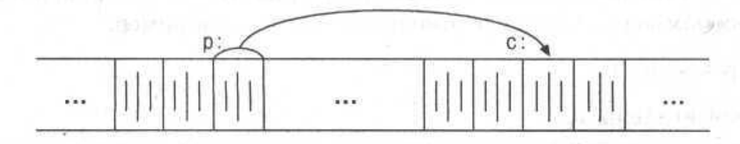
Наряду с goto указатели когда-то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, указывающий на что-нибудь совсем нежелательное. При соблюдении же определенной дисциплины с помощью указателей можно достичь ясности и простоты. Мы попытаемся убедить вас в этом.

Изменения, внесенные стандартом ANSI, связаны в основном с формулированием точных правил, как работать с указателями. Стандарт узаконил накопленный положительный опыт программистов и удачные нововведения разработчиков компиляторов. Кроме того, взамен char* в качестве типа обобщенного указателя предлагается тип void* (указатель на void).

5.1. Указатели и адреса

Начнем с того, что рассмотрим упрощенную схему организации памяти. Память типичной машины представляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа char, двухбайтовые ячейки могут рассматриваться как целое типа short, а четырехбайтовые - как целые типа long. Указатель - это группа

ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если c имеет тип char, а p – указатель на c, то ситуация выглядит следующим образом:



Унарный оператор & выдает адрес объекта, так что инструкция

```
p = &c;
```

присваивает переменной p адрес ячейки c (говорят, что p указывает на c). Оператор & применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Унарный оператор * есть оператор *косвенного доступа.* Примененный к указателю он выдает объект, на который данный указатель указывает. Предположим, что x и y имеют тип int, а ip - указатель на int. Следующие несколько строк придуманы специально для того, чтобы показать, каким образом объявляются указатели и как используются операторы & и *.

```
int x = 1, y = 2, z[10];
int *ip;            /* ip - указатель на int */

ip = &x;            /* теперь ip указывает на x */
y = *ip;            /* y теперь равен 1 */
*ip = 0;            /* x теперь равен 0 */
ip = &z[0];         /* ip теперь указывает на z[0] */
```

Объявления x, y и z нам уже знакомы. Объявление указателя ip

```
int *ip;
```

мы стремились сделать мнемоничным - оно гласит: "выражение *ip имеет тип int". Синтаксис объявления переменной "подстраивается" под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в объявлениях функций. Например, запись

```
double *dp, atof (char *);
```

означает, что выражения *dp и atof(s) имеют тип double, а аргумент функции atof есть указатель на char.

Вы, наверное, заметили, что указателю разрешено указывать только на объекты определенного типа. (Существует одно исключение: "указатель

на void" может указывать на объекты любого типа, но к такому указателю нельзя применять оператор косвенного доступа. Мы вернемся к этому в параграфе 5.11.)

Если ip указывает на x целочисленного типа, то *ip можно использовать в любом месте, где допустимо применение x; например,

```
*ip = *ip+ 10;
```

увеличивает *ip на 10.

Унарные операторы * и & имеют более высокий приоритет, чем арифметические операторы, так что присваивание

```
y = *ip + 1
```

берет то, на что указывает ip, и добавляет к нему 1, а результат присваивает переменной y. Аналогично

```
*ip += 1
```

увеличивает на единицу то, на что указывает ip; те же действия выполняют

```
++*ip
```

и

```
(*ip)++
```

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операторы * и ++ имеют одинаковый приоритет и порядок выполнения - справа налево.

И наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора косвенного доступа. Например, если iq есть другой указатель на int, то

```
iq = ip
```

копирует содержимое ip в iq, чтобы ip и iq указывали на один и тот же объект.

5.2. Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции. В программе сортировки нам понадобилась функция swap, меняющая местами два неупорядоченных элемента. Однако недостаточно написать

```
swap(a, b);
```

где функция swap определена следующим образом:

```
void swap(int x, int y)          /* НЕВЕРНО */
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

Поскольку swap получает лишь *копии* переменных a и b, она не может повлиять на переменные a и b той программы, которая к ней обратилась.

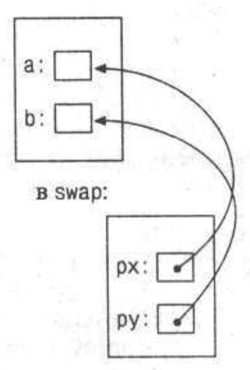
Чтобы получить желаемый эффект, вызывающей программе надо передать *указатели* на те значения, которые должны быть изменены:

```
swap(&a, &b);
```

Так как оператор & получает адрес переменной, &a есть указатель на a. В самой же функции swap параметры должны быть объявлены как указатели, при этом доступ к значениям параметров будет осуществляться косвенно.

```
void swap(int *px, int *py) /* перестановка *px и *py */
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

Графически это выглядит следующим образом:



Аргументы-указатели позволяют функции осуществлять доступ к объектам вызвавшей ее программы и дают возможность изменить эти объекты. Рассмотрим, например, функцию getint, которая осуществляет ввод в свободном формате одного целого числа и его перевод из текстового представления в значение типа int. Функция getint должна возвращать значение полученного числа или сигнализировать значением EOF о конце файла, если входной поток исчерпан. Эти значения должны возвращаться по разным каналам, так как нельзя рассчитывать на то, что полученное в результате перевода число никогда не совпадет с EOF.

Одно из решений состоит в том, чтобы getint выдавала характеристику состояния файла (исчерпан или не исчерпан) в качестве результата, а значение самого числа помещала согласно указателю, переданному ей в виде аргумента. Похожая схема действует и в программе scanf, которую мы рассмотрим в параграфе 7.4.

Показанный ниже цикл заполняет некоторый массив целыми числами, полученными с помощью getint.

```
int n, array[SIZE], getint (int *);
for (n = 0; n < SIZE && getint (&array[n]) != EOF; n++)
    ;
```

Результат каждого очередного обращения к getint посылается в array[n], и n увеличивается на единицу. Заметим, и это существенно, что функции getint передается адрес элемента array[n]. Если этого не сделать, у getint не будет способа вернуть в вызывающую программу переведенное целое число.

В предлагаемом нами варианте функция getint возвращает EOF по концу файла; нуль, если следующие вводимые символы не представляют собою числа; и положительное значение, если введенные символы представляют собой число.

```
#include <ctype.h>

int getch (void);
void ungetch (int);

/* getint: читает следующее целое из ввода в *pn */
int getint(int *pn)
{
  int c, sign;

  while (isspace(c = getch()))
    ;                     /* пропуск символов-разделителей */
```

```
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch (c);          /* не число */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Везде в getint под *рп подразумевается обычная переменная типа int. Функция ungetch вместе с getch (параграф 4.3) включена в программу, чтобы обеспечить возможность отослать назад лишний прочитанный символ.

Упражнение 5.1. Функция getint написана так, что знаки - или +, за которыми не следует цифра, она понимает как "правильное" представление нуля. Скорректируйте программу таким образом, чтобы в подобных случаях она возвращала прочитанный знак назад во ввод.

Упражнение 5.2. Напишите функцию getfloat – аналог getint для чисел с плавающей точкой. Какой тип будет иметь результирующее значение, выдаваемое функцией getfloat?

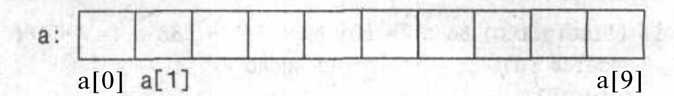
5.3. Указатели и массивы

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя. Вариант с указателями в общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно.

Объявление

```
int a[10];
```

определяет массив а размера 10, т. е. блок из 10 последовательных объектов с именами a[0], a[1],..., a[9].



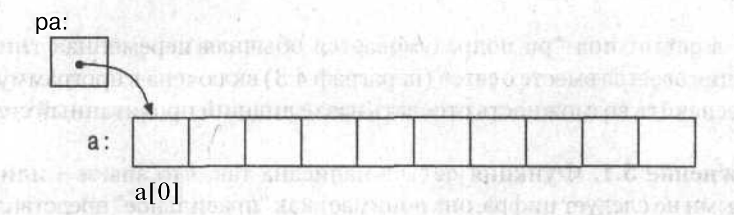
Запись a[i] отсылает нас к i-му элементу массива. Если pa есть указатель на int, т. е. объявлен как

```
int *pa;
```

то в результате присваивания

```
pa = &a[0];
```

pa будет указывать на нулевой элемент a, иначе говоря, pa будет содержать адрес элемента a[0].



Теперь присваивание

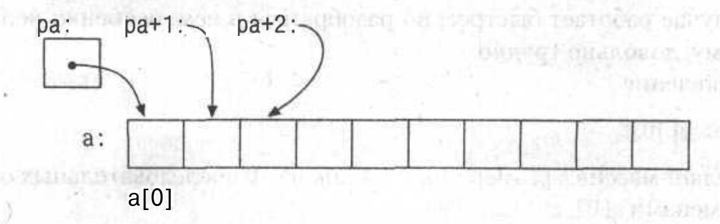
```
x = *pa;
```

будет копировать содержимое a[0] в x.

Если pa указывает на некоторый элемент массива, то pa+1 по определению указывает на следующий элемент, pa+i - на i-й элемент после pa, а pa-i - на i-й элемент перед pa. Таким образом, если pa указывает на a[0], то

```
*(pa+1)
```

есть содержимое a[1], a+i – адрес a[i], а *(pa+i) – содержимое a[i].



Сделанные замечания верны безотносительно к типу и размеру элементов массива a. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы pa+1 указывал на следующий объект, a pa+i - на i-й после pa.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания

```
pa = &a[0];
```

pa и a имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание pa=&a[0] можно также записать в следующем виде:

```
pa = a;
```

Еще более удивительно (по крайней мере на первый взгляд) то, что a[i] можно записать как *(a+i). Вычисляя a[i], Си сразу преобразует его в *(a+i); указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора & записи &a[i] и a+i также будут эквивалентными, т. е. и в том и в другом случае это адрес i-го элемента после a. С другой стороны, если pa - указатель, то его можно использовать с индексом, т. е. запись pa[i] эквивалентна записи *(pa+i). Короче говоря, элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель - это переменная, поэтому можно написать pa=a или pa++. Но имя массива не является переменной, и записи вроде a=pa или a++ не допускаются.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать еще одну версию функции strlen, вычисляющей длину строки.

```
/* strlen: возвращает длину строки */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Так как переменная s - указатель, к ней применима операция ++; s++ не оказывает никакого влияния на строку символов функции, которая обратилась к strlen. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции strlen. Это значит, что все вызовы, такие как:

```
strlen("Здравствуй, мир");  /* строковая константа */
strlen(array);              /* char array[100]; */
strlen(ptr);                /* char *ptr; */
```

правомерны.

Формальные параметры

```
char s[];
```

и

```
char *s;
```

в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что s есть указатель. Если функции в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно - либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если a - массив, то в записях

```
f(&a[2])
```

или

```
f(a+2)
```

функции f передается адрес подмассива, начинающегося с элемента a[2]. Внутри функции f описание параметров может выглядеть как

```
f(int arr[]) { ... }
```

или

```
f(int *arr) { ... }
```

Следовательно, для f тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к нулевому элементу; выражения p[-1], p[-2] и т. д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед p[0]. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

5.4. Адресная арифметика

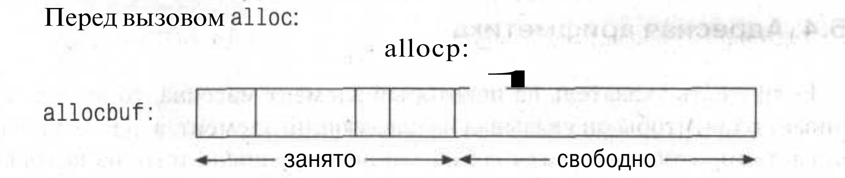
Если p есть указатель на некоторый элемент массива, то p++ увеличивает p так, чтобы он указывал на следующий элемент, а p += i увеличивает его, чтобы он указывал на i-й элемент после того, на который указывал ранее. Эти и подобные конструкции - самые простые примеры арифметики над указателями, называемой также адресной арифметикой.

Си последователен и единообразен в своем подходе к адресной арифметике. Это соединение в одном языке указателей, массивов и адресной арифметики - одна из сильных его сторон. Проиллюстрируем сказанное построением простого распределителя памяти, состоящего из двух программ. Первая, alloc(n), возвращает указатель p на n последовательно расположенных ячеек типа char; программой, обращающейся к alloc, эти ячейки могут быть использованы для запоминания символов. Вторая, afree(p), освобождает память для, возможно, повторной ее утилизации. Простота алгоритма обусловлена предположением, что обращения к afree делаются в обратном порядке по отношению к соответствующим обращениям к alloc. Таким образом, память, с которой работают alloc и afree, является стеком (списком, в основе которого лежит принцип "последним вошел, первым ушел"). В стандартной библиотеке имеются функции malloc и free, которые делают то же самое, только без упомянутых ограничений; в параграфе 8.7 мы покажем, как они выглядят.

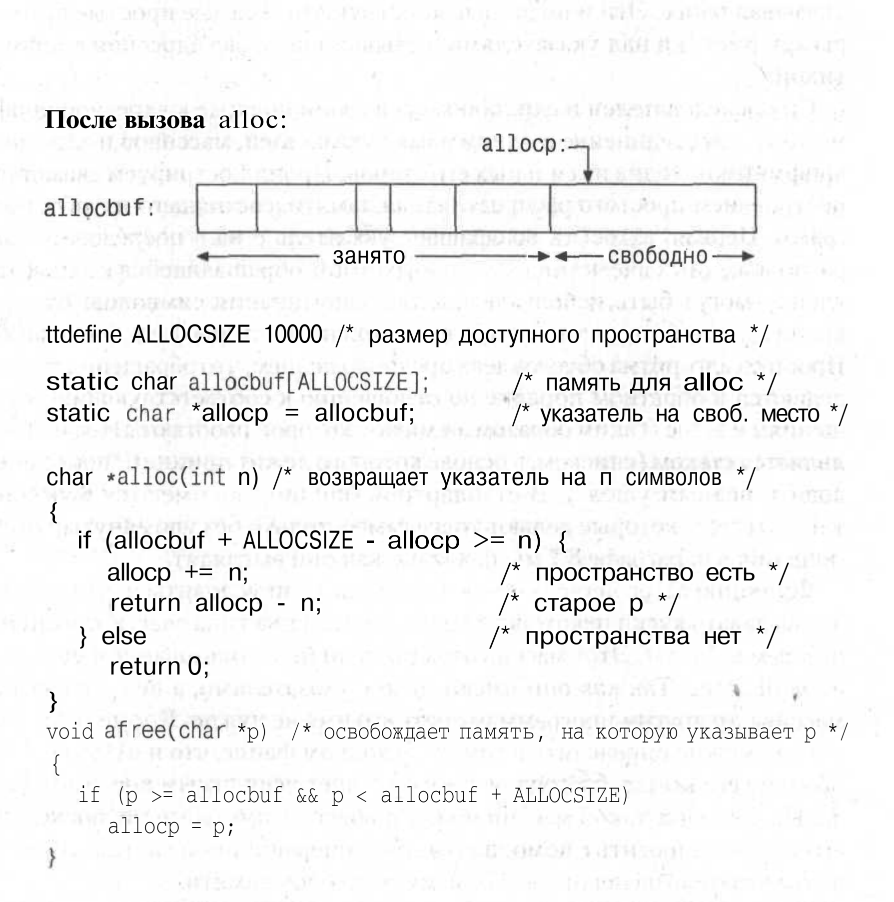
Функцию alloc легче всего реализовать, если условиться, что она будет выдавать куски некоторого большого массива типа char, который мы назовем allocbuf. Этот массив отдадим в личное пользование функциям alloc и afree. Так как они имеют дело с указателями, а не с индексами массива, то другим программам знать его имя не нужно. Кроме того, этот массив можно определить в том же исходном файле, что и alloc и afree, объявив его static, благодаря чему он станет невидимым вне этого файла. На практике такой массив может и вовсе не иметь имени, поскольку его можно запросить с помощью malloc у операционной системы и получить указатель на некоторый безымянный блок памяти.

Естественно, нам нужно знать, сколько элементов массива allocbuf уже занято. Мы введем указатель allocp, который будет указывать на первый свободный элемент. Если запрашивается память для n символов, то alloc возвращает текущее значение allocp (т. е. адрес начала свободного блока) и затем увеличивает его на n, чтобы указатель allocp указывал на следующую свободную область. Если же пространства нет, то alloc выдает нуль. Функция afree[p] просто устанавливает allocp в значение p, если оно не выходит за пределы массива allocbuf.

Перед вызовом alloc:



После вызова alloc:




```
ttdefine ALLOCSIZE 10000 /* размер доступного пространства */

static char allocbuf[ALLOCSIZE];        /* память для alloc */
static char *allocp = allocbuf;         /* указатель на своб. место */

char *alloc(int n) /* возвращает указатель на п символов */
{
   if (allocbuf + ALLOCSIZE - allocp >= n) {
      allocp += n;                      /* пространство есть */
      return allocp - n;                /* старое р */
   } else                               /* пространства нет */
      return 0;
}
void afree(char *p)  /* освобождает память, на которую указывает р */
{
   if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
      allocp = p;
}
```

В общем случае указатель, как и любую другую переменную, можно инициализировать, но только такими осмысленными для него значениями, как нуль или выражение, приводящее к адресу ранее определенных данных соответствующего типа. Объявление

```
static char *allocp = allocbuf;
```

определяет allocp как указатель на char и инициализирует его адресом массива allocbuf, поскольку перед началом работы программы массив

allocbuf пуст. Указанное объявление могло бы иметь и такой вид:

```
static char *allocp = &allocbuf[0];
```

поскольку имя массива и есть адрес его нулевого элемента.

Проверка

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* годится */
```

контролирует, достаточно ли пространства, чтобы удовлетворить запрос на n символов. Если памяти достаточно, то новое значение для allocp должно указывать не далее чем на следующую позицию за последним элементом allocbuf. При выполнении этого требования alloc выдает указатель на начало выделенного блока символов (обратите внимание на объявление типа самой функции). Если требование не выполняется, функция alloc должна выдать какой-то сигнал о том, что памяти не хватает. Си гарантирует, что нуль никогда не будет правильным адресом для данных, поэтому мы будем использовать его в качестве признака аварийного события, в нашем случае нехватки памяти.

Указатели и целые не являются взаимозаменяемыми объектами. Константа нуль - единственное исключение из этого правила: ее можно присвоить указателю, и указатель можно сравнить с нулевой константой. Чтобы показать, что нуль - это специальное значение для указателя, вместо цифры нуль, как правило, записывают NULL - константу, определенную в файле <stdio.h>. С этого момента и мы будем ею пользоваться.

Проверки

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* годится */
```

и

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

демонстрируют несколько важных свойств арифметики с указателями. Во-первых, при соблюдении некоторых правил указатели можно сравнивать.

Если p и q указывают на элементы одного массива, то к ним можно применять операторы отношения ==, !=, <, >= и т. д. Например, отношение вида

```
p < q
```

истинно, если p указывает на более ранний элемент массива, чем q. Любой указатель всегда можно сравнить на равенство и неравенство с нулем. А вот для указателей, не указывающих на элементы одного массива, результат арифметических операций или сравнений не определен. (Существует одно исключение: в арифметике с указателями можно использовать адрес несуществующего "следующего за массивом" элемента,

т. е. адрес того "элемента", который станет последним, если в массив добавить еще один элемент.)

Во-вторых, как вы уже, наверное, заметили, указатели и целые можно складывать и вычитать. Конструкция

```
p + п
```

означает адрес объекта, занимающего л-е место после объекта, на который указывает р. Это справедливо безотносительно к типу объекта, на который указывает р; п автоматически домножается на коэффициент, соответствующий размеру объекта. Информация о размере неявно присутствует в объявлении р. Если, к примеру, int занимает четыре байта, то коэффициент умножения будет равен четырем.

Допускается также вычитание указателей. Например, если р и q указывают на элементы одного массива и p<q, то q-p+1 есть число элементов от р до q включительно. Этим фактом можно воспользоваться при написании еще одной версии strlen:

```
/* strlen: возвращает длину строки s */
int strlen(char *s)
{
   char *p = s;

   while (*p != '\0')
      p++;
   return p - s;
}
```

В своем объявлении р инициализируется значением s, т. е. вначале р указывает на первый символ строки. На каждом шаге цикла while проверяется очередной символ; цикл продолжается до тех пор, пока не встретится '\0'. Каждое продвижение указателя р на следующий символ выполняется инструкцией р++, и разность p-s дает число пройденных символов, т. е. длину строки. (Число символов в строке может быть слишком большим, чтобы хранить его в переменной типа int. Тип ptrdiff_t, достаточный для хранения разности (со знаком) двух указателей, определен в заголовочном файле <stddef.h>. Однако, если быть очень осторожными, нам следовало бы для возвращаемого результата использовать тип size_t, в этом случае наша программа соответствовала бы стандартной библиотечной версии. Тип size_t есть тип беззнакового целого, возвращаемого оператором sizeof.)

Арифметика с указателями учитывает тип: если она имеет дело со значениями float, занимающими больше памяти, чем char, и р - указатель на float, то р++ продвинет р на следующее значение float. Это значит, что

другую версию alloc, которая имеет дело с элементами типа float, а не char, можно получить простой заменой в alloc и afree всех char на float. Все операции с указателями будут автоматически откорректированы в соответствии с размером объектов, на которые указывают указатели.

Можно производить следующие операции с указателями: присваивание значения указателя другому указателю того же типа, сложение и вычитание указателя и целого, вычитание и сравнение двух указателей, указывающих на элементы одного и того же массива, а также присваивание указателю нуля и сравнение указателя с нулем. Других операций с указателями производить не допускается. Нельзя складывать два указателя, перемножать их, делить, сдвигать, выделять разряды; указатель нельзя складывать со значением типа float или double; указателю одного типа нельзя даже присвоить указатель другого типа, не выполнив предварительно операции приведения (исключение составляют лишь указатели типа void *).

5.5. Символьные указатели функции

Строковая константа, написанная в виде

```
"Я строка"
```

есть массив символов. Во внутреннем представлении этот массив заканчивается нулевым символом '\0', по которому программа может найти конец строки. Число занятых ячеек памяти на одну больше, чем количество символов, помещенных между двойными кавычками.

Чаще всего строковые константы используются в качестве аргументов функций, как, например, в

```
printf("здравствуй, мир\n");
```

Когда такая символьная строка появляется в программе, доступ к ней осуществляется через символьный указатель; printf получает указатель на начало массива символов. Точнее, доступ к строковой константе осуществляется через указатель на ее первый элемент.

Строковые константы нужны не только в качестве аргументов функций. Если, например, переменную pmessage объявить как

```
char *pmessage
```

то присваивание

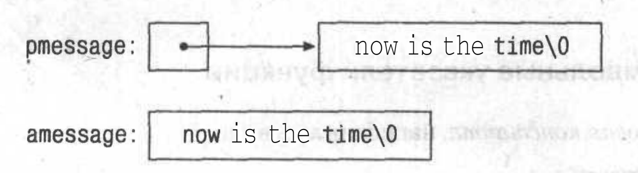
```
pmessage = "now is the time";
```

поместит в нее указатель на символьный массив, при этом сама строка *не* копируется, копируется лишь указатель на нее. Операции для работы со строкой как с единым целым в Си не предусмотрены.

Существует важное различие между следующими определениями:

```
char amessage[] = "now is the time";     /* массив */
char *pmessage  = "now is the time";     /* указатель */
```

amessage - это массив, имеющий такой объем, что в нем как раз помещается указанная последовательность символов и '\0'. Отдельные символы внутри массива могут изменяться, но amessage всегда указывает на одно и то же место памяти. В противоположность ему pmessage есть указатель, инициализированный так, чтобы указывать на строковую константу. А значение указателя можно изменить, и тогда последний будет указывать на что-либо другое. Кроме того, результат будет неопределен, если вы попытаетесь изменить содержимое константы.



Дополнительные моменты, связанные с указателями и массивами, проиллюстрируем на несколько видоизмененных вариантах двух полезных программ, взятых нами из стандартной библиотеки. Первая из них, функция strcpy(s, t), копирует строку t в строку s. Хотелось бы написать прямо s=t, но такой оператор копирует указатель, а не символы. Чтобы копировать символы, нам нужно организовать цикл. Первый вариант strcpy, с использованием массива, имеет следующий вид:

```
/* strcpy: копирует t в s; вариант с индексируемым массивом*/
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0' )
        i++;
}
```

Для сравнения приведем версию strcpy с указателями:

```
/* strcpy: копирует t в s: версия 1 (с указателями) */
void strcpy(char *s, char *t)
```

```
{
  while ((*s = *t) != '\0' ) {
    s++;
    t++;
  }
}
```

Поскольку передаются лишь копии значений аргументов, strcpy может свободно пользоваться параметрами s и t как своими локальными переменными. Они должным образом инициализированы указателями, которые продвигаются каждый раз на следующий символ в каждом из массивов до тех пор, пока в копируемой строке t не встретится '\0'.

На практике strcpy так не пишут. Опытный программист предпочтет более короткую запись:

```
/* strcpy: копирует t в s; версия 2 (с указателями) */
void strcpy(char *s, char *t)
{
  while ((*s++ = *t++) != '\0'}
    ;
}
```

Приращение s и t здесь осуществляется в управляющей части цикла. Значением *t++ является символ, на который указывает переменная t перед тем, как ее значение будет увеличено; постфиксный оператор ++ не изменяет указатель t, пока не будет взят символ, на который он указывает. То же в отношении s: сначала символ запомнится в позиции, на которую указывает старое значение s, и лишь после этого значение переменной s увеличится. Пересылаемый символ является одновременно и значением, которое сравнивается с '\0'. В итоге копируются все символы, включая и заключительный символ '\0'.

Заметив, что сравнение с '\0' здесь лишнее (поскольку в Си ненулевое значение выражения в условии трактуется и как его истинность), мы можем сделать еще одно и последнее сокращение текста программы:

```
/* strcpy: копирует t в s; версия 3 (с указателями) */
void strcpy(char *s, char *t)
{
  while (*s++ = *t++)
    ;
}
```

Хотя на первый взгляд то, что мы получили, выглядит загадочно, все же такая запись значительно удобнее, и следует освоить ее, поскольку в Си-программах вы будете с ней часто встречаться.

Что касается функции strcpy из стандартной библиотеки <string.h>, то она возвращает в качестве своего результата еще и указатель на новую копию строки.

Вторая программа, которую мы здесь рассмотрим, это strcmp(s,t). Она сравнивает символы строк s и t и возвращает отрицательное, нулевое или положительное значение, если строка s соответственно лексикографически меньше, равна или больше, чем строка t. Результат получается вычитанием первых несовпадающих символов из s и t.

```
/* strcmp: выдает < 0 при s < t, 0 при s == t, > 0 при s > t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

Та же программа с использованием указателей выглядит так:

```
/* strcmp: выдает < 0 при s < t, 0 при s == t, > 0 при s > t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Поскольку операторы ++ и -- могут быть или префиксными, или постфиксными, встречаются (хотя и не так часто) другие их сочетания с оператором *. Например:

```
*--p
```

уменьшит p прежде, чем по этому указателю будет получен символ. Например, следующие два выражения:

```
*p++ = val;  /* поместить val в стек */
val = *--p;  /* взять из стека значение и поместить в val */
```

являются стандартными для посылки в стек и взятия из стека (см. параграф 4.3.).

Объявления функций, упомянутых в этом параграфе, а также ряда других стандартных функций, работающих со строками, содержатся в заголовочном файле <string.h>.

Упражнение 5.3. Используя указатели, напишите функцию strcat, которую мы рассматривали в главе 2 (функция strcat(s,t) копирует строку t в конец строки s).

Упражнение 5.4. Напишите функцию strend(s,t), которая выдает 1, если строка t расположена в конце строки s, и нуль в противном случае.

Упражнение 5.5. Напишите варианты библиотечных функций strncpy, strncat и strncmp, которые оперируют с первыми символами своих аргументов, число которых не превышает n. Например, strncpy (t,s,n) копирует не более n символов t в s. Полные описания этих функций содержатся в приложении В.

Упражнение 5.6. Отберите подходящие программы из предыдущих глав и упражнений и перепишите их, используя вместо индексирования указатели. Подойдут, в частности, программы getline (главы 1 и 4), atoi, itoa и их варианты (главы 2, 3 и 4), reverse (глава 3), а также strindex и getop (глава 4).

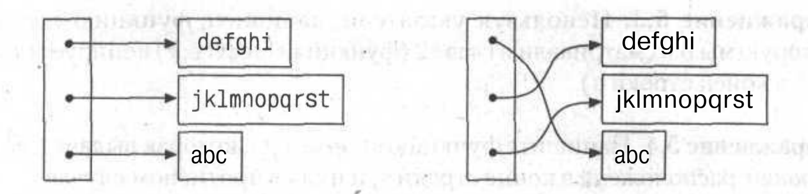
5.6. Массивы указателей, указатели на указатели

Как и любые другие переменные, указатели можно группировать в массивы. Для иллюстрации этого напишем программу, сортирующую в алфавитном порядке текстовые строки; это будет упрощенный вариант программы sort системы UNIX.

В главе 3 мы привели функцию сортировки по Шеллу, которая упорядочивает массив целых, а в главе 4 улучшили ее, повысив быстродействие. Те же алгоритмы используются и здесь, однако теперь они будут обрабатывать текстовые строки, которые могут иметь разную длину и сравнение или перемещение которых невозможно выполнить за одну операцию. Нам необходимо выбрать некоторое представление данных, которое бы позволило удобно и эффективно работать с текстовыми строками произвольной длины.

Для этого воспользуемся массивом указателей на начала строк. Поскольку строки в памяти расположены вплотную друг к другу, к каждой отдельной строке доступ просто осуществлять через указатель на ее первый символ. Сами указатели можно организовать в виде массива. Одна

из возможностей сравнить две строки - передать указатели на них функции strcmp. Чтобы поменять местами строки, достаточно будет поменять местами в массиве их указатели (а не сами строки).



Здесь снимаются сразу две проблемы: одна - связанная со сложностью управления памятью, а вторая - с большими накладными расходами при перестановках самих строк.

Процесс сортировки распадается на три этапа:

чтение всех строк из ввода
сортировка введенных строк
печать их по порядку

Как обычно, выделим функции, соответствующие естественному делению задачи, и напишем главную программу main, управляющую этими функциями. Отложим на время реализацию этапа сортировки и сосредоточимся на структуре данных и вводе-выводе.

Программа ввода должна прочитать и запомнить символы всех строк, а также построить массив указателей на строки. Она, кроме того, должна подсчитать число введенных строк - эта информация понадобится для сортировки и печати. Так как функция ввода может работать только с конечным числом строк, то, если их введено слишком много, она будет выдавать некоторое значение, которое никогда не совпадет с количеством строк, например -1.

Программа вывода занимается только тем, что печатает строки, причем в том порядке, в котором расположены указатели на них в массиве.

```
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000      /* максимальное число строк */

char *lineptr[MAXLINES];  /* указатели на строки */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
```

```
void qsort(char *lineptr[], int left, int right);

/* сортировка строк */
main()
{
  int nlines;           /* количество прочитанных строк */

  if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
    qsort(lineptr, 0, nlines-1);
    writelines(lineptr, nlines);
    return 0;
  } else {
    printf("ошибка: слишком много строк\n");
    return 1;
  }
}

tfdefine MAXLEN 1000      /* максимальная длина строки */
int getline(char *, int);
char *alloc(int);

/* readlines: чтение строк */
int readlines(char *lineptr[], int maxlines)
{
  int len, nlines;
  char *p, line[MAXLEN];

  nlines = 0;
  while ((len = getline(line, MAXLEN)) > 0)
    if (nlines >= maxlines || (p = alloc(len)) == NULL)
      return -1;
    else {
      line[len-1] = '\0'; /* убираем символ \n */
      strcpy(p, line);
      lineptr[nlines++] = p;
    }
  return nlines;
}

/* writelines: печать строк */
void writelines(char *lineptr[], int nlines)
{
  int i;
```

```
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Функция getline **взята из параграфа 1.9.**

Основное новшество здесь - объявление lineptr:

```
char *lineptr[MAXLINES]
```

в котором сообщается, что lineptr есть массив из MAXLINES элементов, каждый из которых представляет собой указатель на char. Иначе говоря, lineptr[i] - указатель на символ, а *lineptr[i] - символ, на который он указывает (первый символ i-й строки текста).

Так как lineptr - имя массива, его можно трактовать как указатель, т. е. так же, как мы это делали в предыдущих примерах, и writelines переписать следующим образом:

```
/* writelines: печать строк */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Вначале *lineptr указывает на первую строку; каждое приращение указателя приводит к тому, что *lineptr указывает на следующую строку, и делается это до тех пор, пока nlines не станет нулем.

Теперь, когда мы разобрались с вводом и выводом, можно приступить к сортировке. Быструю сортировку, описанную в главе 4, надо несколько модифицировать: нужно изменить объявления, а операцию сравнения заменить обращением к strcmp. Алгоритм остался тем же, и это дает нам определенную уверенность в его правильности.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)  /* ничего не делается, если в массиве */
        return;         /* менее двух элементов */
    swap(v, left, (left+ right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
```

```
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Небольшие поправки требуются и в программе перестановки.

```
/* swap: поменять местами v[i] и v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Так как каждый элемент массива v (т. е. lineptr) является указателем на символ, temp должен иметь тот же тип, что и v - тогда можно будет осуществлять пересылки между temp и элементами v.

Упражнение 5.7. Напишите новую версию readlines, которая запоминала бы строки в массиве, определенном в main, а не запрашивала память посредством программы alloc. Насколько быстрее эта программа?

5.7. Многомерные массивы

В Си имеется возможность задавать прямоугольные многомерные массивы, правда, на практике по сравнению с массивами указателей они используются значительно реже. В этом параграфе мы продемонстрируем некоторые их свойства.

Рассмотрим задачу перевода даты "день-месяц" в "день года" и обратно. Например, 1 марта - это 60-й день невисокосного или 61-й день високосного года. Определим две функции для этих преобразований: функция day_of_year будет преобразовывать месяц и день в день года, а month_day - день года в месяц и день. Поскольку последняя функция вычисляет два значения, аргументы месяц и день будут указателями. Так вызов

```
month_day(1988, 60, &m, &d)
```

присваивает переменной m значение 2, а d - 29 (29 февраля).

Нашим функциям нужна одна и та же информация, а именно таблица, содержащая числа дней каждого месяца. Так как для високосного и невисокосного годов эти таблицы будут различаться, проще иметь две отдельные строки в двумерном массиве, чем во время вычислений отслеживать особый случай с февралем. Массив и функции, выполняющие преобразования, имеют следующий вид:

```
static char daytab[2][13] = {
  {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: определяет день года по месяцу и дню */
int day_of_year(int year, int month, int day)
{
  int i, leap;

  leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
  for (i = 1; i < month; i++)
    day += daytab[leap][i];
  return day;
}
/* month_day: определяет месяц и день по дню года */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
  int i, leap;

  leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
  for (i = 1; yearday > daytab[leap][i]; i++)
    yearday -= daytab[leap][i];
  *pmonth = i;
  *pday = yearday;
}
```

Напоминаем, что арифметическое значение логического выражения (например выражения, с помощью которого вычислялось leap) равно либо нулю (ложь), либо единице (истина), так что мы можем использовать его как индекс в массиве daytab.

Массив daytab должен быть внешним по отношению к обеим функциям day_of_year и month_day, так как он нужен и той и другой. Мы сделали его типа char, чтобы проиллюстрировать законность применения типа char для малых целых без знака.

Массив daytab - это первый массив из числа двумерных, с которыми мы еще не имели дела. Строго говоря, в Си двумерный массив рассматривается как одномерный массив, каждый элемент которого - также массив. Поэтому индексирование изображается так:

```
daytab[i][j]     /* [строка] [столбец] */
```

а не **так:**

```
daytab[i,j]      /* НЕВЕРНО */
```

Особенность двумерного массива в Си заключается лишь в форме записи, в остальном его можно трактовать почти так же, как в других языках. Элементы запоминаются строками, следовательно, при переборе их в том порядке, как они расположены в памяти, чаще будет изменяться самый правый индекс.

Массив инициализируется списком начальных значений, заключенным в фигурные скобки; каждая строка двумерного массива инициализируется соответствующим подсписком. Нулевой столбец добавлен в начало daytab лишь для того, чтобы индексы, которыми мы будем пользоваться, совпадали с естественными номерами месяцев от 1 до 12. Экономить пару ячеек памяти здесь нет никакого смысла, а программа, в которой уже не надо корректировать индекс, выглядит более ясной.

Если двумерный массив передается функции в качестве аргумента, то объявление соответствующего ему параметра должно содержать количество столбцов; количество строк в данном случае несущественно, поскольку, как и прежде, функции будет передан указатель на массив строк, каждая из которых есть массив из 13 значений типа int. В нашем частном случае мы имеем указатель на объекты, являющиеся массивами из 13 значений типа int. Таким образом, если массив daytab передается некоторой функции f, то эту функцию можно было бы определить следующим образом:

```
f(int daytab[2][13]) { ... }
```

Вместо этого можно записать

```
f(int daytab[][13]) { ... }
```

поскольку число строк здесь не имеет значения, или

```
f(int (*daytab)[13]) { ... }
```

Последняя запись объявляет, что параметр есть указатель на массив из 13 значений типа int. Скобки здесь необходимы, так как квадратные скобки [ ] имеют более высокий приоритет, чем *. Без скобок объявление

```
int *daytab[13]
```

определяет массив из 13 указателей на char. В более общем случае только

первое измерение (соответствующее первому индексу) можно не задавать, все другие специфицировать необходимо.

В параграфе 5.12 мы продолжим рассмотрение сложных объявлений.

Упражнение 5.8. В функциях day_of_year и month_day нет никаких проверок правильности вводимых дат. Устраните этот недостаток.

5.8. Инициализация массивов указателей

Напишем функцию month_name(n), которая возвращает указатель на строку символов, содержащий название n-го месяца. Эта функция идеальна для демонстрации использования статического массива. Функция month_name имеет в своем личном распоряжении массив строк, на одну из которых она и возвращает указатель. Ниже покажем, как инициализируется этот массив имен.

Синтаксис задания начальных значений аналогичен синтаксису предыдущих инициализаций:

```
/* month_name: возвращает имя n-го месяца */
char *month_name(int n)
{
   static char *name[] = {
      "Неверный месяц",
      "Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",
      "Июль", "Август", "Сентябрь",
      "Октябрь", "Ноябрь", "Декабрь"
   } ;
   return (a < 1 || n > 12) ? name[0] : name[n];
}
```

Объявление name массивом указателей на символы такое же, как и объявление lineptr в программе сортировки. Инициализатором служит список строк, каждой из которых соответствует определенное место в массиве. Символы i-й строки где-то размещены, и указатель на них запоминается в name[i]. Так как размер массива name не специфицирован, компилятор вычислит его по количеству заданных начальных значений.

5.9. Указатели против многомерных массивов

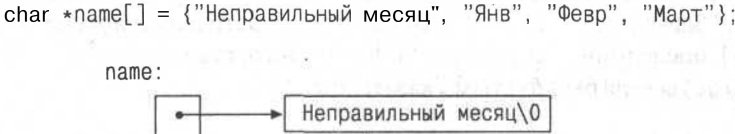
Начинающие программировать на Си иногда не понимают, в чем разница между двумерным массивом и массивом указателей вроде name из приведенного примера. Для двух следующих определений:

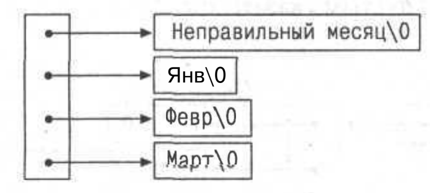
```
int a[10][20];
int *b[10];
```

записи a[3][4] и b[3][4] будут синтаксически правильным обращением к некоторому значению типа int. Однако только a является истинно двумерным массивом: для двухсот элементов типа int будет выделена память, а вычисление смещения элемента a[*строка*][*столбец*] от начала массива будет вестись по формуле 20 x *строка* + *столбец,* учитывающей его прямоугольную природу. Для b же определено только 10 указателей, причем без инициализации. Инициализация должна задаваться явно - либо статически, либо в программе. Предположим, что каждый элемент b указывает на двадцатиэлементный массив, в результате где-то будут выделены пространство, в котором разместятся 200 значений типа int, и еще 10 ячеек для указателей. Важное преимущество массива указателей в том, что строки такого массива могут иметь разные длины. Таким образом, каждый элемент массива b не обязательно указывает на двадцатиэлементный вектор; один может указывать на два элемента, другой - на пятьдесят, а некоторые и вовсе могут ни на что не указывать.

Наши рассуждения здесь касались целых значений, однако чаще массивы указателей используются для работы со строками символов, различающимися по длине, как это было в функции month_name. Сравните определение массива указателей и соответствующий ему рисунок:

```
char *name[] = {"Неправильный месяц", "Янв", "Февр", "Март"};
```





с объявлением и рисунком для двумерного массива:

```
char aname[][15] = {"Неправ. месяц", "Янв", "Февр", "Март"};
```

aname:

Неправ.месяц\0	Янв\0	Февр\0	Март\0
0	15	30	45

Упражнение 5.9. Перепишите программы day_of_year и month_day, используя вместо индексов указатели.

5.10. Аргументы командной строки

В операционной среде, обеспечивающей поддержку Си, имеется возможность передать аргументы или параметры запускаемой программе с помощью командной строки. В момент вызова main получает два аргумента. В первом, обычно называемом argc (сокращение от *argument count)*, стоит количество аргументов, задаваемых в командной строке. Второй, argv (от *argument vector)*, является указателем на массив символьных строк, содержащих сами аргументы. Для работы с этими строками обычно используются указатели нескольких уровней.

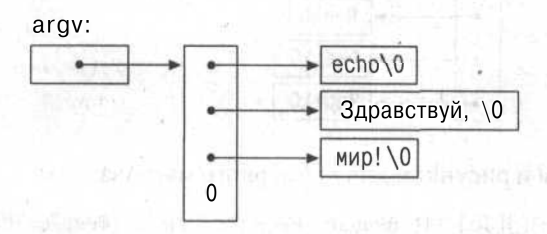
Простейший пример - программа echo ("эхо"), которая печатает аргументы своей командной строки в одной строчке, отделяя их друг от друга пробелами. Так, команда

```
echo Здравствуй, мир!
```

напечатает

```
Здравствуй, мир!
```

По соглашению argv[0] есть имя вызываемой программы, так что значение argc никогда не бывает меньше 1. Если argc равен 1, то в командной строке после имени программы никаких аргументов нет. В нашем примере argc равен 3, и соответственно argv[0], argv[1] и argv[2] суть строки "echo", "Здравствуй," и "мир!". Первый необязательный аргумент - это argv[1], последний - argv[argc-1]. Кроме того, стандарт требует, чтобы argv[argc] всегда был пустым указателем.



Первая версия программы echo трактует argv как массив символьных указателей.

```
tfinclude <stdio.h>

/* эхо аргументов командной строки: версия 1 */
main(int argc, char *argv[])
```

```
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Так как argv - это указатель на массив указателей, мы можем работать с ним как с указателем, а не как с индексируемым массивом. Следующая программа основана на приращении argv, он приращивается так, что его значение в каждый отдельный момент указывает на очередной указатель на char; перебор указателей заканчивается, когда исчерпан argc.

```
#include <stdio.h>

/* эхо аргументов командной строки; версия 2 */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Аргумент argv – указатель на начало массива строк аргументов. Использование в ++argv префиксного оператора ++ приведет к тому, что первым будет напечатан argv[1], а не argv[0]. Каждое очередное приращение указателя дает нам следующий аргумент, на который указывает *argv. В это же время значение argc уменьшается на 1, и, когда оно станет нулем, все аргументы будут напечатаны.

Инструкцию printf можно было бы написать и так:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Как видим, формат в printf тоже может быть выражением.

В качестве второго примера возьмем программу поиска образца, рассмотренную в параграфе 4.1, и несколько усовершенствуем ее. Если вы помните, образец для поиска мы "вмонтировали" глубоко в программу, а это, очевидно, не лучшее решение. Построим нашу программу по аналогии с grep из UNIXа, т. е. так, чтобы образец для поиска задавался первым аргументом в командной строке.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: печать строк с образцом, заданным 1-м аргументом */
main(int argc, char *argv[])
{
  char line [MAXLINE],
  int found = 0;
  if (argc != 2)
    printf("Используйте в find образец\n");
  else
    while (getline(line, MAXLINE) > 0)
      if (strstr(line, argv[1]) != NULL) {
        printf ("%s",line);
        found++;
      }
  return found;
}
```

Стандартная функция strstr(s,t) возвращает указатель на первую встретившуюся строку t в строке s или NULL, если таковой в s не встретилось. Функция объявлена в заголовочном файле <string.h>.

Эту модель можно развивать и дальше, чтобы проиллюстрировать другие конструкции с указателями. Предположим, что мы вводим еще два необязательных аргумента. Один из них предписывает печатать все строки, *кроме* тех, в которых встречается образец; второй - перед каждой выводимой строкой печатать ее порядковый номер.

По общему соглашению для Си-программ в системе UNIX знак минус перед аргументом вводит необязательный признак или параметр. Так, если -x служит признаком слова "кроме", которое изменяет задание на противоположное, а -n указывает на потребность в нумерации строк, то команда

find -x -n ***образец***

напечатает все строки, в которых не найден указанный образец, и, кроме того, перед каждой строкой укажет ее номер.

Необязательные аргументы разрешается располагать в любом порядке, при этом лучше, чтобы остальная часть программы не зависела от числа представленных аргументов. Кроме того, пользователю было бы удобно, если бы он мог комбинировать необязательные аргументы, например так:

find -nx ***образец***

А теперь запишем нашу программу.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: печать строк образцами из 1-го аргумента */
main(int angc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;
    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf( "find: неверный параметр %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Используйте: find -x -n образец\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

Перед получением очередного аргумента argc уменьшается на 1, а argv "перемещается" на следующий аргумент. После завершения цикла при отсутствии ошибок argc содержит количество еще не обработанных аргументов, а argv указывает на первый из них. Таким образом, argc должен быть равен 1, а *argv указывать на образец. Заметим, что *++argv является указателем на аргумент-строку, а (*++argv)[0] - его первым символом, на который можно сослаться и другим способом:

```
**++argv.
```

Поскольку оператор индексирования [ ] имеет более высокий приоритет, чем * и ++, круглые скобки здесь обязательны, без них выражение трактовалось бы так же, как *++(argv[0]). Именно такое выражение мы применим во внутреннем цикле, где просматриваются символы конкретного аргумента. Во внутреннем цикле выражение *++argv[0] приращивает указатель argv[0].

Потребность в более сложных выражениях для указателей возникает не так уж часто. Но если такое случится, то разбивая процесс вычисления указателя на два или три шага, вы облегчите восприятие этого выражения.

Упражнение 5.10. Напишите программу expr, интерпретирующую обратную польскую запись выражения, задаваемого командной строкой, в которой каждый оператор и операнд представлены отдельным аргументом. Например,

```
expr 2 3 4 + *
```

вычисляется так же, как выражение 2 x (3 + 4).

Упражнение 5.11. Усовершенствуйте программы entab и detab (см. упражнения 1.20 и 1.21) таким образом, чтобы через аргументы можно было задавать список "стопов" табуляции.

Упражнение 5.12. Расширьте возможности entab и detab таким образом, чтобы при обращении вида

entab *-m* *+n*

"стопы" табуляции начинались с *m*-й позиции и выполнялись через каждые *n* позиций. Разработайте удобный для пользователя вариант поведения программы по умолчанию (когда нет никаких аргументов).

Упражнение 5.13. Напишите программу tail, печатающую *n* последних введенных строк. По умолчанию значение *n* равно 10, но при желании *n* можно задать с помощью аргумента. Обращение вида

tail *-n*

печатает *n* последних строк. Программа должна вести себя осмысленно при любых входных данных и любом значении *n*. Напишите программу так, чтобы наилучшим образом использовать память; запоминание строк организуйте, как в программе сортировки, описанной в параграфе 5.6, а не на основе двумерного массива с фиксированным размером строки.

5.11. Указатели на функции

В Си сама функция не является переменной, но можно определить указатель на функцию и работать с ним, как с обычной переменной: присваивать, размещать в массиве, передавать в качестве параметра функции, возвращать как результат из функции и т. д. Для иллюстрации этих возможностей воспользуемся программой сортировки, которая уже встречалась в настоящей главе. Изменим ее так, чтобы при задании необязательного аргумента -n вводимые строки упорядочивались по их числовому значению, а не в лексикографическом порядке.

Сортировка, как правило, распадается на три части: на сравнение, определяющее упорядоченность пары объектов; перестановку, меняющую местами пару объектов, и сортирующий алгоритм, который осуществляет сравнения и перестановки до тех пор, пока все объекты не будут упорядочены. Алгоритм сортировки не зависит от операций сравнения и перестановки, так что передавая ему в качестве параметров различные функции сравнения и перестановки, его можно настроить на различные критерии сортировки.

Лексикографическое сравнение двух строк выполняется функцией strcmp (мы уже использовали эту функцию в ранее рассмотренной программе сортировки); нам также потребуется программа numcmp, сравнивающая две строки как числовые значения и возвращающая результат сравнения в том же виде, в каком его выдает strcmp. Эти функции объявляются перед main, а указатель на одну из них передается функции qsort. Чтобы сосредоточиться на главном, мы упростили себе задачу, отказавшись от анализа возможных ошибок при задании аргументов.

```
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000      /* максимальное число строк */
char *lineptr[MAXLINES];  /* указатели на строки текста */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
```

```
void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* сортировка строк */
main(int argc, char *argv[])
{
    int nlines;       /* количество прочитанных строк */
    int numeric = 0; /* 1, если сорт. по числ. знач. */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
          (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("введено слишком много строк\n");
        return 1;
    }
}
```

В обращениях к функциям qsort, strcmp и numcmp их имена трактуются как адреса этих функций, поэтому оператор & перед ними не нужен, как он не был нужен и перед именем массива.

Мы написали qsort так, чтобы она могла обрабатывать данные любого типа, а не только строки символов. Как видно из прототипа, функция qsort в качестве своих аргументов ожидает массив указателей, два целых значения и функцию с двумя аргументами-указателями. В качестве аргументов-указателей заданы указатели обобщенного типа void *. Любой указатель можно привести к типу void * и обратно без потери информации, поэтому мы можем обратиться к qsort, предварительно преобразовав аргументы в void*. Внутри функции сравнения ее аргументы будут приведены к нужному ей типу. На самом деле эти преобразования никакого влияния на представления аргументов не оказывают, они лишь обеспечивают согласованность типов для компилятора.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
```

```
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right)      /* ничего не делается, если */
        return;             /* в массиве менее двух элементов */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Повнимательней приглядимся к объявлениям. Четвертый параметр функции qsort:

```
int (*comp)(void *, void *)
```

сообщает, что comp - это указатель на функцию, которая имеет два аргумента-указателя и выдает результат типа int.

Использование comp в строке

```
if ((*comp)(v(il, v[left]) < 0)
```

согласуется с объявлением "comp - это указатель на функцию", и, следовательно, *comp - это функция, а

```
(*comp)(v[i], v[left])
```

- обращение к ней. Скобки здесь нужны, чтобы обеспечить правильную трактовку объявления; без них объявление

```
int *comp(void *, void *)       /* НЕВЕРНО */
```

говорило бы, что comp - это функция, возвращающая указатель на int, а это совсем не то, что требуется.

Мы уже рассматривали функцию strcmp, сравнивающую две строки. Ниже приведена функция numcmp, которая сравнивает две строки, рассматривая их как числа; предварительно они переводятся в числовые значения функцией atof.

```
#include <stdlib.h>

/* numcmp: сравнивает s1 и s2 как числа */
```

```
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Функция swap, меняющая местами два указателя, идентична той, что мы привели ранее в этой главе за исключением того, что объявления указателей заменены на void*.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Программу сортировки можно дополнить и множеством других возможностей; реализовать некоторые из них предлагается в качестве упражнений.

Упражнение 5.14. Модифицируйте программу сортировки, чтобы она реагировала на параметр -r, указывающий, что объекты нужно сортировать в обратном порядке, т. е. в порядке убывания. Обеспечьте, чтобы -r работал и вместе с -n.

Упражнение 5.15. Введите в программу необязательный параметр -f, задание которого делало бы неразличимыми символы нижнего и верхнего регистров (например, a и A должны оказаться при сравнении равными).

Упражнение 5.16. Предусмотрите в программе необязательный параметр -d, который заставит программу при сравнении учитывать только буквы, цифры и пробелы. Организуйте программу таким образом, чтобы этот параметр мог работать вместе с параметром -f.

Упражнение 5.17. Реализуйте в программе возможность работы с полями: возможность сортировки по полям внутри строк. Для каждого поля предусмотрите свой набор параметров. Предметный указатель этой книги[1] упорядочивался с параметрами: -df для терминов и -n для номеров страниц.

5.12. Сложные объявления

Иногда Си ругают за синтаксис объявлений, особенно тех, которые содержат в себе указатели на функции. Таким синтаксис получился в результате нашей попытки сделать похожими объявления объектов и их использование. В простых случаях этот синтаксис хорош, однако в сложных ситуациях он вызывает затруднения, поскольку объявления перенасыщены скобками и их невозможно читать слева направо. Проблему иллюстрирует различие следующих двух объявлений:

```
int  *f();          /* f: функция, возвращающая ук-ль на int */
int (*pf)();        /* pf: ук-ль на ф-цию, возвращающую int */
```

Приоритет префиксного оператора * ниже, чем приоритет (), поэтому во втором случае скобки необходимы.

Хотя на практике по-настоящему сложные объявления встречаются редко, все же важно знать, как их понимать, а если потребуется, и как их конструировать.

Укажем хороший способ: объявления можно синтезировать, двигаясь небольшими шагами с помощью typedef; этот способ рассмотрен в параграфе 6.7. В настоящем параграфе на примере двух программ, осуществляющих преобразования правильных Си-объявлений в соответствующие им словесные описания и обратно, мы демонстрируем иной способ конструирования объявлений. Словесное описание читается слева направо.

Первая программа, dcl, - более сложная. Она преобразует Си-объявления в словесные описания так, как показано в следующих примерах:

```
char **argv
   argv: указ. на указ. на char
int (*daytab)[13]
   daytab: указ. на массив[13] из int
int (*daytab)[13]
   daytab: массив[13] из указ. на int
void *comp()
```

[1] Имеется в виду оригинал книги на английском языке. — *Примеч. пер.*

```
    comp: функц. возвр. указ. на void
void (*comp)()
    comp: указ. на функц. возвр. void
char (*(*x())[])()
    x: функц. возвр. указ. на массив[] из указ. на функц.
        возвр. char
char (*(*x[3])())[5]
    x: массив[3] из указ. на функц. возвр. указ.
            на массив[5] из char
```

Функция dcl в своей работе использует грамматику, специфицирующую объявитель. Эта грамматика строго изложена в параграфе 8.5 приложения А, а в упрощенном виде записывается так:

объявитель: необязательные * ***собственно-объявитель***
собственно-объявитель: *имя*
(***объявитель***)
собственно-объявитель()
собственно-объявитель [необязательный *размер*]

Говоря простым языком, ***объявитель*** есть *собственно-объявитель,* перед которым может стоять * (т. е. одна или несколько звездочек), где *собственно-объявитель* есть *имя,* или *объявитель* в скобках, или *собственно-объявитель* с последующей парой скобок, или *собственно-объявитель* с последующей парой квадратных скобок, внутри которых может быть помещен *размер.*

Эту грамматику можно использовать для грамматического разбора объявлений. Рассмотрим, например, такой объявитель:

```
(*pfa[])()
```

Имя pfa будет классифицировано как *имя* и, следовательно, как *собственно-объявитель.* Затем pfa[ ] будет распознано как *собственно-объявитель,* а *pfa[ ] - как ***объявитель*** и, следовательно, (*pfa[ ]) есть *собственно-объявитель.* Далее, (*pfa[ ])() есть *собственно-объявитель* и, таким образом, ***объявитель.*** Этот грамматический разбор можно проиллюстрировать деревом разбора, приведенным на следующей странице (где *собственно-объявитель* обозначен более коротко, а именно ***собств.-объяв.***).

Сердцевиной программы обработки объявителя является пара функций dcl и dirdcl, осуществляющих грамматический разбор объявления согласно приведенной грамматике. Поскольку грамматика определена рекурсивно, эти функции обращаются друг к другу рекурсивно, по мере распознавания отдельных частей объявления. Метод, примененный в обсуждаемой программе для грамматического разбора, называется рекурсивным спуском.

(* pfa []) ()

имя

собств.-объяв.

собств.-объяв.

объяв.

собств.-объяв.

собств.-объяв.

объяв.

```
/* dcl: разбор объявителя */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; )   /* подсчет звездочек */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " указ. на");
}
/* dirdcl: разбор собственно объявителя */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {                 /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("ошибка: пропущена )\n");
    } else if (tokentype == NAME)           /* имя переменной */
        strcpy(name, token);
    else
        printf("ошибка: должно быть name или (dcl)\n");
    while((type = gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " функц. возвр.");
        else {
            strcat(out, " массив");
```

```
            strcat(out, token);
            strcat(out, " из");
        }
    }
```

Приведенные программы служат только иллюстративным целям и не вполне надежны. Что касается dcl, то ее возможности существенно ограничены. Она может работать только с простыми типами вроде char и int и не справляется с типами аргументов в функциях и с квалификаторами вроде const. Лишние пробелы для нее опасны. Она не предпринимает никаких мер по выходу из ошибочной ситуации, и поэтому неправильные описания также ей противопоказаны. Устранение этих недостатков мы оставляем для упражнений.

Ниже приведены глобальные переменные и главная программа main.

```
#include <stdio.h>
«include <string.h>
#include <ctype.h>

#define MAXTOKEN 100

enum {NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;                  /* тип последней лексемы */
char token[MAXTOKEN];           /* текст последней лексемы */
char name[MAXTOKEN],            /* имя */
char datatype[MAXTOKEN];        /* тип = char, int и т.д. */
char out[1000];                 /* выдаваемый текст */

main() /* преобразование объявления в словесное описание */
{
    while (gettoken() != EOF) {   /* 1-я лексема в строке */
        strcpy(datatype, token);  /* это тип данных */
        out[0] = '\0';
        dcl(); /* разбор остальной части строки */
        if (tokentype != '\n')
            printf("синтаксическая ошибка\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

Функция gettoken пропускает пробелы и табуляции и затем получает следующую лексему из ввода; "лексема" *(token)* - это имя, или пара круглых скобок, или пара квадратных скобок (быть может, с помещенным в них числом), или любой другой единичный символ.

```
int gettoken(void)      /* возвращает следующую лексему */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token,"()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

Функции getch и ungetch были рассмотрены в главе 4.

Обратное преобразование реализуется легче, особенно если не придавать значения тому, что будут генерироваться лишние скобки. Программа undcl превращает фразу вроде "x есть функция, возвращающая указатель на массив указателей на функции, возвращающие char", которую мы будем представлять в виде

```
x () * [] * () char
```

в объявление

```
char (*(*x())[])()
```

Такой сокращенный входной синтаксис позволяет повторно пользоваться функцией gettoken. Функция undcl использует те же самые внешние переменные, что и dcl.

```
/* undcl: преобразует словесное описание в объявление */
main()
{
  int  type;
  char temp[MAXTOKEN];

  while (gettoken() != EOF) {
    strcpy(out, token);
    while ((type = gettoken()) != '\n')
      if (type == PARENS || type == BRACKETS)
        strcat(qut, token);
      else if (type == '*') {
        sprintf(temp, "(*%s)",out);
        strcpy(out, temp);
      } else if (type == NAME) {
        sprintf(temp, "%s %s", token, out);
        strcpy(out, temp);
      } else
        printf( "неверный элемент %s в фразе\n", token);
    printf("%s\n", out);
  }
  return 0;
}
```

Упражнение 5.18. Видоизмените dcl таким образом, чтобы она обрабатывала ошибки во входной информации.

Упражнение 5.19. Модифицируйте undcl так, чтобы она не генерировала лишних скобок.

Упражнение 5.20. Расширьте возможности dcl, чтобы dcl обрабатывала объявления с типами аргументов функции, квалификаторами вроде const и т. п.

Глава 6

Структуры

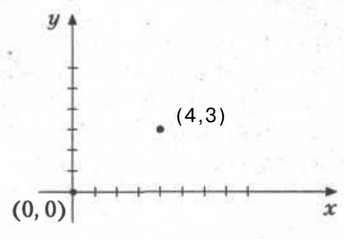
Структура - это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем. (В некоторых языках, в частности в Паскале, структуры называются записями.) Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Традиционный пример структуры - строка платежной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и т. д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент (фамилии, имени и отчества); аналогично адрес, и даже зарплата. Другой пример (более типичный для Си) - из области графики: точка есть пара координат, прямоугольник есть пара точек и т. д.

Главные изменения, внесенные стандартом ANSI в отношении структур, - это введение для них операции присваивания. Структуры могут копироваться, над ними могут выполняться операции присваивания, их можно передавать функциям в качестве аргументов, а функции могут возвращать их в качестве результатов. В большинстве компиляторов уже давно реализованы эти возможности, но теперь они точно оговорены стандартом. Для автоматических структур и массивов теперь также допускается инициализация.

6.1. Основные сведения о структурах

Сконструируем несколько графических структур. В качестве основного объекта выступает точка с координатами x и y целого типа.



Указанные две компоненты можно поместить в структуру, объявленную, например, следующим образом:

```
struct point {
    int x;
    int y;
};
```

Объявление структуры начинается с ключевого слова struct и содержит список объявлений, заключенный в фигурные скобки. За словом struct может следовать имя, называемое ***тегом***[1] *структуры,* (point в нашем случае). Тег дает название структуре данного вида и далее может служить кратким обозначением той части объявления, которая заключена в фигурные скобки.

Перечисленные в структуре переменные называются *элементами (members)*[2]. Имена элементов и тегов без каких-либо коллизий могут совпадать с именами обычных переменных (т. е. не элементов), так как они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в разных структурах, хотя, если следовать хорошему стилю программирования, лучше одинаковые имена давать только близким по смыслу объектам.

Объявление структуры определяет тип. За правой фигурной скобкой, закрывающей список элементов, могут следовать переменные точно так же, как они могут быть указаны после названия любого базового типа. Таким образом, выражение

```
struct { ... } x, y, z;
```

с точки зрения синтаксиса аналогично выражению

```
int x, y, z;
```

в том смысле, что и то и другое объявляет x, y и z переменными указанного типа; и то и другое приведет к выделению памяти соответствующего размера.

[1] От английского слова *tag* — ярлык, этикетка. - *Примеч. пер.*

[2] В некоторых изданиях (в том числе во 2-м издании на русском языке этой книги) *structure members* переводится как *члены структуры.* *Примеч. ред.*

Объявление структуры, не содержащей списка переменных, не резервирует памяти; оно просто описывает шаблон, или образец структуры. Однако если структура имеет тег, то этим тегом далее можно пользоваться при определении структурных объектов. Например, с помощью заданного выше описания структуры point строка

```
struct point pt;
```

определяет структурную переменную pt типа struct point. Структурную переменную при ее определении можно инициализировать, формируя список инициализаторов ее элементов в виде константных выражений:

```
struct point maxpt = { 320, 200 };
```

Инициализировать автоматические структуры можно также присваиванием или обращением к функции, возвращающей структуру соответствующего типа.

Доступ к отдельному элементу структуры осуществляется посредством конструкции вида:

имя-структуры. *элемент*

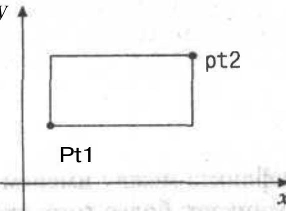
Оператор доступа к элементу структуры . соединяет имя структуры и имя элемента. Чтобы напечатать, например, координаты точки pt, годится следующее обращение к printf:

```
printf("%d,%d", pt.x, pt.y);
```

Другой пример: чтобы вычислить расстояние от начала координат (0,0) до pt, можно написать

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структуры могут быть вложены друг в друга. Одно из возможных представлений прямоугольника - это пара точек на углах одной из его диагоналей:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Структура rect содержит две структуры point. Если мы объявим screen как

```
struct rect screen;
```

то

```
screen.pt1.x
```

обращается к координате x точки pt1 из screen.

6.2. Структуры и функции

Единственно возможные операции над структурами - это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к ее элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значенийй. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений ее элементов; автоматическую структуру также можно инициализировать присваиванием.

Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, makepoint, получает два целых значения и возвращает структуру point.

```
/* makepoint: формирует точку по компонентам x и y */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Заметим: никакого конфликта между именем аргумента и именем элемента структуры не возникает; более того, сходство подчеркивает родство обозначаемых им объектов.

Теперь с помощью makepoint можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

Следующий шаг состоит в определении ряда функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

```
/* addpoint: сложение двух точек */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Здесь оба аргумента и возвращаемое значение - структуры. Мы увеличиваем компоненты прямо в p1 и не используем для этого временной переменной, чтобы подчеркнуть, что структурные параметры передаются по значению так же, как и любые другие.

В качестве другого примера приведем функцию ptinrect, которая проверяет: находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

```
/* ptinrect: возвращает 1, если p в r, и 0 в противном случае */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Здесь предполагается, что прямоугольник представлен в стандартном виде, т. е. координаты точки pt1 меньше соответствующих координат точки pt2. Следующая функция гарантирует получение прямоугольника в каноническом виде.

```
#define min(a, b) ((a) < (D) ? (a) : (b))
«define max(a, b) ((a) > (b) ? (a) : (b))
```

```
/* canonrect: канонизация координат прямоугольника */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

Если функции передается большая структура, то, чем копировать ее целиком, эффективнее передать указатель на нее. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Объявление

```
struct point *pp;
```

сообщает, что pp - это указатель на структуру типа struct point. Если pp указывает на структуру point, то *pp - это сама структура, а (*pp).x и (*pp).y - ее элементы. Используя указатель pp, мы могли бы написать

```
struct point origin, *pp;

pp = &origin;
printf("origin: (%d,%d)\n", (*pp).x, (*pp).y);
```

Скобки в (*pp).x необходимы, поскольку приоритет оператора . выше, чем приоритет *. Выражение *pp.x будет проинтерпретировано как *(pp.x), что неверно, поскольку pp.x не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана еще одна, более короткая форма записи. Если p — указатель на структуру, то

p -> *элемент-структуры*

есть ее отдельный элемент. (Оператор -> состоит из знака -, за которым сразу следует знак >.) Поэтому printf можно переписать в виде

```
printf("origin: (%d,%d)\n", pp->x, pp->y);
```

Операторы . и -> выполняются слева направо. Таким образом, при наличииобъявления

```
struct rect r. *rp = &r;
```

следующие четыре выражения будут эквивалентны:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Операторы доступа к элементам структуры . и -> вместе с операторами вызова функции () и индексации массива [ ] занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задано объявление

```
struct {
   int len;
   char *str;
} *p;
```

то

```
++p->len
```

увеличит на 1 значение элемента структуры len, а не указатель p, поскольку в этом выражении как бы неявно присутствуют скобки: ++(p->len). Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в (++p)->len, прежде чем взять значение len, программа прирастит указатель p. В (p++)->len указатель p увеличится после того, как будет взято значение len (в последнем случае скобки не обязательны).

По тем же правилам *p->str обозначает содержимое объекта, на который указывает str; *p->str++ прирастит указатель str после получения значения объекта, на который он указывал (как и в выражении *s++); (*p->str)++ увеличит значение объекта, на который указывает str; *p++->str увеличит p после получения того, на что указывает str.

6.3. Массивы структур

Рассмотрим программу, определяющую число вхождений каждого ключевого слова в текст Си-программы. Нам нужно уметь хранить ключевые слова в виде массива строк и счетчики ключевых слов в виде массива целых. Один из возможных вариантов - это иметь два параллельных массива:

```
char *keyword[NKEYS];
int  keycount[NKEYS];
```

Однако именно тот факт, что они параллельны, подсказывает нам другую организацию хранения - через массив структур. Каждое ключевое

слово можно описать парой характеристик

```
char *word;
int  count;
```

Такие пары составляют массив. Объявление

```
struct key {
   char *word;
   int  count;
} keytab[NKEYS];
```

объявляет структуру типа key и определяет массив keytab, каждый элемент которого является структурой этого типа и которому где-то будет выделена память. Это же можно записать и по-другому:

```
struct key {
   char *word;
   int  count;
};
struct key keytab[NKEYS];
```

Так как keytab содержит постоянный набор имен, его легче всего сделать внешним массивом и инициализировать один раз в момент определения. Инициализация структур аналогична ранее демонстрировавшимся инициализациям — за определением следует список инициализаторов, заключенный в фигурные скобки:

```
struct key {
   char *word;
   int  count;
} keytab[] = {
   "auto", 0,
   "break", 0,
   "case", 0,
   "char", 0,
   "const", 0,
   "continue", 0,
   "default", 0,
   Л ... V
   "unsigned", 0,
   "void", 0,
   "volatile", 0,
   "while", 0
};
```

Инициализаторы задаются парами, чтобы соответствовать конфигурации структуры. Строго говоря, пару инициализаторов для каждой отдельной структуры следовало бы заключить в фигурные скобки, как, например, в

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

Однако когда инициализаторы - простые константы или строки символов и все они имеются в наличии, во внутренних скобках нет необходимости. Число элементов массива keytab будет вычислено по количеству инициализаторов, поскольку они представлены полностью, а внутри квадратных скобок "[ ]" ничего не задано.

Программа подсчета ключевых слов начинается с определения keytab. Программа main читает ввод, многократно обращаясь к функции getword и получая на каждом ее вызове очередное слово. Каждое слово ищется в keytab. Для этого используется функция бинарного поиска, которую мы написали в главе 3. Список ключевых слов должен быть упорядочен в алфавитном порядке.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* подсчет ключевых слов Си */
main()
{
    int n;
    char word[MAXWORD];

    while(getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
}
```

```
/* binsearch: найти слово в tab[0]...tab[n - 1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high)/2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Чуть позже мы рассмотрим функцию getword, а сейчас нам достаточно знать, что при каждом ее вызове получается очередное слово, которое запоминается в массиве, заданном первым аргументом.

NKEYS - количество ключевых слов в keytab. Хотя мы могли бы подсчитать число таких слов вручную, гораздо легче и безопасней сделать это с помощью машины, особенно если список ключевых слов может быть изменен. Одно из возможных решений - поместить в конец списка инициализаторов пустой указатель (NULL) и затем перебирать в цикле элементы keytab, пока не встретится концевой элемент.

Но возможно и более простое решение. Поскольку размер массива полностью определен во время компиляции и равен произведению количества элементов массива на размер его отдельного элемента, число элементов массива можно вычислить по формуле

размер keytab / *размер* struct key

В Си имеется унарный оператор sizeof, который работает во время компиляции. Его можно применять для вычисления размера любого объекта. Выражения

sizeof *объект*

и

sizeof (*имя типа*)

выдают целые значения, равные размеру указанного объекта или типа

в байтах. (Строго говоря, sizeof выдает беззнаковое целое, тип которого size_t определен в заголовочном файле <stddef.h>.) Что касается объекта, то это может быть переменная, массив или структура. В качестве имени типа может выступать имя базового типа (int, double...) или имя производного типа, например структуры или указателя.

В нашем случае, чтобы вычислить количество ключевых слов, размер массива надо поделить на размер одного элемента. Указанное вычисление используется в инструкции #define для установки значения NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Этот же результат можно получить другим способом - поделить размер массива на размер какого-то его конкретного элемента:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

Преимущество такого рода записей в том, что их не надо корректировать при изменении типа.

Поскольку препроцессор не обращает внимания на имена типов, оператор sizeof нельзя применять в #if. Но в #define выражение препроцессором не вычисляется, так что предложенная нами запись допустима.

Теперь поговорим о функции getword. Мы написали getword в несколько более общем виде, чем требуется для нашей программы, но она от этого не стала заметно сложнее. Функция getword берет из входного потока следующее "слово". Под словом понимается цепочка букв-цифр, начинающаяся с буквы, или отдельный символ, отличный от символа-разделителя. В случае конца файла функция возвращает EOF, в остальных случаях ее значением является код первого символа слова или сам символ, если это не буква.

```
/* getword: принимает следующее слово или символ из ввода */
int getword (char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
```

```
for ( ; --lim > 0; w++)
    if (!isalnum(*w = getch())) {
        ungetch(*w);
        break;
    }
*w = '\0';
return word[0];
}
```

Функция getword обращается к getch и ungetch, которые мы написали в главе 4. По завершении набора букв-цифр оказывается, что getword взяла лишний символ. Обращение к ungetch позволяет вернуть его назад во входной поток. В getword используются также isspace - для пропуска символов-разделителей, isalpha - для идентификации букв и isalnum - для распознавания букв-цифр. Все они описаны в стандартном заголовочном файле <ctype.h>.

Упражнение 6.1. Наша версия getword не обрабатывает должным образом знак подчеркивания, строковые константы, комментарии и управляющие строки препроцессора. Напишите более совершенный вариант программы.

6.4. Указатели на структуры

Для иллюстрации некоторых моментов, касающихся указателей на структуры и массивов структур, перепишем программу подсчета ключевых слов, пользуясь для получения элементов массива вместо индексов указателями.

Внешнее объявление массива keytab остается без изменения, а main и binsearch нужно модифицировать.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* подсчет ключевых слов Си: версия с указателями */
main()
{
    char word[MAXWORD];
    struct key *p;
```

```
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p = binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: найти слово word в tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high - low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Некоторые детали этой программы требуют пояснений. Во-первых, описание функции binsearch должно отражать тот факт, что она возвращает указатель на struct key, а не целое; это объявлено как в прототипе функции, так и в функции binsearch. Если binsearch находит слово, то она выдает указатель на него, в противном случае она возвращает NULL. Во-вторых, к элементам keytab доступ в нашей программе осуществляется через указатели. Это потребовало значительных изменений в binsearch. Инициализаторами для low и high теперь служат указатели на начало и на место сразу после конца массива. Вычисление положения среднего элемента с помощью формулы

```
mid = (low + high) / 2        /* НЕВЕРНО */
```

не годится, поскольку указатели нельзя складывать. Однако к ним можно

применить операцию вычитания, и так как high-low есть число элементов, присваивание

```
mid = low + (high-low) / 2
```

превратит mid в указатель на элемент, лежащий посередине между low и high.

Самое важное при переходе на новый вариант программы - сделать так, чтобы не генерировались неправильные указатели и не было попыток обратиться за пределы массива. Проблема в том, что и &tab[-1], и &tab[n] находятся вне границ массива. Первый адрес определенно неверен, нельзя также осуществить доступ и по второму адресу. По правилам языка, однако, гарантируется, что адрес ячейки памяти, следующей сразу за концом массива (т. е. &tab[n]), в арифметике с указателями воспринимается правильно.

В главной программе main мы написали

```
for (p - keytab; p < keytab + NKEYS; p++)
```

Если p - это указатель на структуру, то при выполнении операций с p учитывается размер структуры. Поэтому p++ увеличит p на такую величину, чтобы выйти на следующий структурный элемент массива, а проверка условия вовремя остановит цикл.

Не следует, однако, полагать, что размер структуры равен сумме размеров ее элементов. Вследствие выравнивания объектов разной длины в структуре могут появляться безымянные "дыры". Например, если переменная типа char занимает один байт, а int - четыре байта, то для структуры

```
struct {
    char c;
    int i;
};
```

может потребоваться восемь байтов, а не пять. Оператор sizeof возвращает правильное значение.

Наконец, несколько слов относительно формата программы. Если функция возвращает значение сложного типа, как, например, в нашем случае она возвращает указатель на структуру:

```
struct key *binsearch(char *word, struct key *tab, int n)
```

то "высмотреть" имя функции оказывается совсем не просто. В подобных случаях иногда пишут так:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Какой форме отдать предпочтение - дело вкуса. Выберите ту, которая больше всего вам нравится, и придерживайтесь ее.

6.5. Структуры со ссылками на себя

Предположим, что мы хотим решить более общую задачу - написать программу, подсчитывающую частоту встречаемости для *любых слов* входного потока. Так как список слов заранее не известен, мы не можем предварительно упорядочить его и применить бинарный поиск. Было бы неразумно пользоваться и линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет - в этом случае программа работала бы слишком медленно. (Более точная оценка: время работы такой программы пропорционально квадрату количества слов.) Как можно организовать данные, чтобы эффективно справиться со списком произвольных слов?

Один из способов - постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Делать это передвижкой слов в линейном массиве не следует, - хотя бы потому, что указанная процедура тоже слишком долгая. Вместо этого мы воспользуемся структурой данных, называемой *бинарным деревом*.

В дереве на каждое отдельное слово предусмотрен "узел", который содержит:

- указатель на текст слова;
- счетчик числа встречаемости;
- указатель на левый сыновний узел;
- указатель на правый сыновний узел.

У каждого узла может быть один или два сына, или узел вообще может не иметь сыновей.

Узлы в дереве располагаются так, что по отношению к любому узлу левое поддерево содержит только те слова, которые лексикографически меньше, чем слово данного узла, а правое - слова, которые больше него. Вот как выглядит дерево, построенное для фразы "now is the time for all good men to come to the aid of their party" ("настало время всем добрым людям помочь своей партии"), по завершении процесса, в котором для каждого нового слова в него добавлялся новый узел:

```
                  now
                /     \
             is         the
            /  \       /    \
         for    men  of      time
        /   \          \     /   \
      all   good      party their  to
     /   \
   aid   come
```

Чтобы определить, помещено ли уже в дерево вновь поступившее слово, начинают с корня, сравнивая это слово со словом из корневого узла. Если они совпали, то ответ на вопрос — положительный. Если новое слово меньше слова из дерева, то поиск продолжается в левом поддереве, если больше, то — в правом. Если же в выбранном направлении поддерева не оказалось, то этого слова в дереве нет, а пустующая позиция, говорящая об отсутствии поддерева, как раз и есть то место, куда нужно "подвесить" узел с новым словом. Описанный процесс по сути рекурсивен, так как поиск в любом узле использует результат поиска в одном из своих сыновних узлов. В соответствии с этим для добавления узла и печати дерева здесь наиболее естественно применить рекурсивные функции.

Вернемся к описанию узла, которое удобно представить в виде структуры с четырьмя компонентами:

```
    struct tnode {          /* узел дерева */
      char *word;           /* указатель на текст */
      int  count;           /* число вхождений */
      struct tnode *left;   /* левый сын */
      struct tnode *right;  /* правый сын */
    } ;
```

Приведенное рекурсивное определение узла может показаться рискованным, но оно правильное. Структура не может включать саму себя, но ведь

```
struct tnode *left;
```

объявляет left как указатель на tnode, а не сам tnode.

Иногда возникает потребность во взаимоссылающихся структурах: двух структурах, ссылающихся друг на друга. Прием, позволяющий справиться с этой задачей, демонстрируется следующим фрагментом:

```
struct t {
    ...
    struct s *p; /* p указывает на s */
};
struct s {
    ...
    struct t *q; /* q указывает на t */
} ;
```

Вся программа удивительно мала - правда, она использует вспомогательные программы вроде getword, уже написанные нами. Главная программа читает слова с помощью getword и вставляет их в дерево посредством addtree.

```
#include <stdio.h>
tfinclude <ctype. h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
• void treeprint(struct tnode *);
int getword(char *, int);

/* подсчет частоты встречаемости слов */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword (word, MAXWORD) != EOF)
        if (lsalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

Функция addtree рекурсивна. Первое слово функция main помещает на верхний уровень дерева (корень дерева). Каждое вновь поступившее слово сравнивается со словом узла и "погружается" или в левое, или в правое поддерево с помощью рекурсивного обращения к addtree. Через некоторое время это слово обязательно либо совпадет с каким-нибудь из имеющихся в дереве слов (в этом случае к счетчику будет добавлена 1), либо программа встретит пустую позицию, что послужит сигналом для создания нового узла и добавления его к дереву. Создание нового узла сопровождается тем, что addtree возвращает на него указатель, который вставляется в узел родителя.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree: добавляет узел со словом w в p или ниже него */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {        /* слово встречается впервые */
```

```
        p = talloc();           /* создается новый узел */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;             /* это слово уже встречалось */
    else if (cond < 0)          /* меньше корня левого поддерева */
        p->left = addtree(p->left, w);
    else                        /* больше корня правого поддерева */
        p->right = addtree(p->right, w);
    return p;
}
```

Память для нового узла запрашивается с помощью программы talloc, которая возвращает указатель на свободное пространство, достаточное для хранения одного узла дерева, а копирование нового слова в отдельное место памяти осуществляется с помощью strdup. (Мы рассмотрим эти программы чуть позже.) В тот (и только в тот) момент, когда к дереву подвешивается новый узел, происходит инициализация счетчика и обнуление указателей на сыновей. Мы опустили (что неразумно) контроль ошибок, который должен выполняться при получении значений от strdup и talloc.

Функция treeprint печатает дерево в лексикографическом порядке; для каждого узла она печатает его левое поддерево (все слова, которые меньше слова данного узла), затем само слово и, наконец, правое поддерево (слова, которые больше слова данного узла).

```
/* treeprint: упорядоченная печать дерева p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Если вы не уверены, что досконально разобрались в том, как работает рекурсия, "проиграйте" действия treeprint на дереве, приведенном выше.

Практическое замечание: если дерево "несбалансировано" (что бывает, когда слова поступают не в случайном порядке), то время работы программы может сильно возрасти. Худший вариант, когда слова уже упорядочены; в этом случае затраты на вычисления будут такими же, как при

линейном поиске. Существуют обобщения бинарного дерева, которые не страдают этим недостатком, но здесь мы их не описываем.

Прежде чем завершить обсуждение этого примера, сделаем краткое отступление от темы и поговорим о механизме запроса памяти. Очевидно, хотелось бы иметь всего лишь одну функцию, выделяющую память, даже если эта память предназначается для разного рода объектов. Но если одна и та же функция обеспечивает память, скажем, и для указателей на `char`, и для указателей на `struct tnode`, то возникают два вопроса. Первый: как справиться с требованием большинства машин, в которых объекты определенного типа должны быть выровнены (например, int часто должны размещаться, начиная с четных адресов)? И второе: как объявить функцию-распределитель памяти, которая вынуждена в качестве результата возвращать указатели разных типов?

Вообще говоря, требования, касающиеся выравнивания, можно легко выполнить за счет некоторого перерасхода памяти. Однако для этого возвращаемый указатель должен быть таким, чтобы удовлетворялись любые ограничения, связанные с выравниванием. Функция `alloc`, описанная в главе 5, не гарантирует нам любое конкретное выравнивание, поэтому мы будем пользоваться стандартной библиотечной функцией `malloc`, которая это делает. В главе 8 мы покажем один из способов ее реализации.

Вопрос об объявлении типа таких функций, как malloc, является камнем преткновения в любом языке с жесткой проверкой типов. В Си вопрос решается естественным образом: malloc объявляется как функция, которая возвращает указатель на `void`. Полученный указатель затем явно приводится к желаемому типу[1]. Описания malloc и связанных с ней функций находятся в стандартном заголовочном файле `<stdlib.h>`. Таким образом, функцию talloc можно записать так:

```
«include <stdlib.h>

/* talloc: создает tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

[1] Замечание о приведении типа величины, возвращаемой функцией `malloc`, нужно переписать. Пример корректен и работает, но совет является спорным в контексте стандартов ANSI/ISO 1988-1989 г. На самом деле это не обязательно (при условии что приведение `void*` к `ALMOSTANYTYPE*` выполняется автоматически) и возможно даже опасно, если malloc или ее заместитель не может быть объявлен как **функций,** возвращающая `void*`. Явное приведение типа может скрыть случайную ошибку. В другие времена (до появления стандарта ANSI) приведение считалось обязательным, что также справедливо и для C++. — *Примеч. авт.*

Функция strdup просто копирует строку, указанную в аргументе, в место, полученное с помощью malloc:

```
char *strdup(char *s)              /* делает дубликат s */
{
  char *p;

  p = (char *) malloc(strlen(s)+1);   /* +1 для '\0' */
  if (p != NULL)
    strcpy(p, s);
  return p;
}
```

Функция malloc возвращает NULL, если свободного пространства нет; strdup возвращает это же значение, оставляя заботу о выходе из ошибочной ситуации вызывающей программе.

Память, полученную с помощью malloc, можно освободить для повторного использования, обратившись к функции free (см. главы 7 и 8).

Упражнение 6.2. Напишите программу, которая читает текст Си-программы и печатает в алфавитном порядке все группы имен переменных, в которых совпадают первые 6 символов, но последующие в чем-то различаются. Не обрабатывайте внутренности закавыченных строк и комментариев. Число 6 сделайте параметром, задаваемым в командной строке.

Упражнение 6.3. Напишите программу печати таблицы "перекрестных ссылок", которая будет печатать все слова документа и указывать для каждого из них номера строк, где оно встретилось. Программа должна игнорировать "шумовые" слова, такие как "и", "или" и пр.

Упражнение 6.4. Напишите программу, которая печатает весь набор различных слов, образующих входной поток, в порядке возрастания частоты их встречаемости. Перед каждым словом должно быть указано число вхождений.

6.6. Просмотр таблиц

В этом параграфе, чтобы проиллюстрировать новые аспекты применения структур, мы напишем ядро пакета программ, осуществляющих вставку элементов в таблицы и их поиск внутри таблиц. Этот пакет - типичный набор программ, с помощью которых работают с таблицами имен

в любом макропроцессоре или компиляторе. Рассмотрим, например, инструкцию #define. Когда встречается строка вида

```
#define IN 1
```

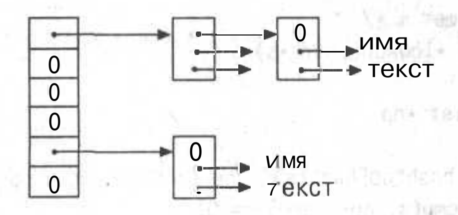
имя IN и замещающий его текст 1 должны запоминаться в таблице. Если затем имя IN встретится в инструкции, например в

```
state = IN;
```

оно должно быть заменено на 1.

Существуют две программы, манипулирующие с именами и замещающими их текстами. Это install(s, t), которая записывает имя s и замещающий его текст t в таблицу, где s и t - строки, и lookup(s), осуществляющая поиск s в таблице и возвращающая указатель на место, где имя s было найдено, или NULL, если s в таблице не оказалось.

Алгоритм основан на хэш-поиске: поступающее имя свертывается в неотрицательное число (хэш-код), которое затем используется в качестве индекса в массиве указателей. Каждый элемент этого массива является указателем на начало связанного списка блоков, описывающих имена с данным хэш-кодом. Если элемент массива равен NULL, это значит, что имен с соответствующим хэш-кодом нет.



Блок в списке - это структура, содержащая указатели на имя, на замещающий текст и на следующий блок в списке; значение NULL в указателе на следующий блок означает конец списка.

```
struct nlist {              /* элемент таблицы */
    struct nlist *next;     /* указатель на следующий элемент */
    char *name;             /* определенное имя */
    char *defn;             /* замещающий текст */
};
```

А вот как записывается определение массива указателей:

```
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE]; /* таблица указателей */
```

Функция хэширования, используемая в lookup и install, суммирует коды символов в строке и в качестве результата выдает остаток от деления полученной суммы на размер массива указателей. Это не самая лучшая функция хэширования, но достаточно лаконичная и эффективная.

```
/* hash: получает хэш-код для строки s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Беззнаковая арифметика гарантирует, что хэш-код будет неотрицательным.

Хэширование порождает стартовый индекс для массива hashtab; если соответствующая строка в таблице есть, она может быть обнаружена только в списке блоков, на начало которого указывает элемент массива hashtab с этим индексом. Поиск осуществляется с помощью lookup. Если lookup находит элемент с заданной строкой, то возвращает указатель на нее, если не находит, то возвращает NULL.

```
/* lookup: ищет s */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;    /* нашли */
    return NULL;          /* не нашли */
}
```

В for-цикле функции lookup для просмотра списка используется стандартная конструкция

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

Функция install обращается к lookup, чтобы определить, имеется ли уже вставляемое имя. Если это так, то старое определение будет заменено новым. В противном случае будет образован новый элемент. Если запрос памяти для нового элемента не может быть удовлетворен, функция install выдает NULL.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: заносит имя и текст (name, defn) в таблицу */
struct nlist *install(char *name, char *defn)
{
  struct nlist *np;
  unsignedhashval;

  if ((np = lookup(name)) == NULL) {     /* не найден */
    np = (struct nlist *) malloc(sizeof(*np));
    if (np == NULL || (np->name = strdup(name)) == NULL)
      return NULL;
    hashval = hash(name);
    np->next = hashtab[hashval];
    hashtab[hashval] = np;
  } else                            /* уже имеется */
    free((void *) np->defn);    /* освобождаем прежний defn */
  if ((np->defn = strdup(defn)) == NULL)
    return NULL;
  return np;
}
```

Упражнение 6.5. Напишите функцию undef, удаляющую имя и определение из таблицы, организация которой поддерживается функциями lookup и install.

Упражнение 6.6. Реализуйте простую версию tfdefine-процессора (без аргументов), которая использовала бы программы этого параграфа и годилась бы для Си-программ. Вам могут помочь программы getch и ungetch.

6.7. Средство typedef

Язык Си предоставляет средство, называемое typedef, которое позволяет давать типам данных новые имена. Например, объявление

```
typedef int Length;
```

делает имя Length синонимом int. С этого момента тип Length можно применять в объявлениях, в операторе приведения и т. д. точно так же, как тип int:

```
Length len, maxlen;
Length *lengths[];
```

Аналогично объявление

```
typedef char *String;
```

делает String синонимом char *, т. е. указателем на char, и правомерным будет, например, следующее его использование:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Заметим, что объявляемый в typedef тип стоит на месте имени переменной в обычном объявлении, а не сразу за словом typedef. С точки зрения синтаксиса слово typedef напоминает класс памяти - extern, static и т. д. Имена типов записаны с заглавных букв для того, чтобы они выделялись.

Для демонстрации более сложных примеров применения typedef воспользуемся этим средством при задании узлов деревьев, с которыми мы уже встречались в данной главе.

```
typedef struct tnode *Treeptr;

typedef struct tnode { /* узел дерева: */
   char *word;            /* указатель на текст */
   int  count;            /* число вхождений */
   Treeptr left;          /* левый сын */
   Treeptr right;         /* правый сын */
} Treenode;
```

В результате создаются два новых названия типов: Treenode (структура) и Treeptr (указатель на структуру). Теперь программу talloc можно записать в следующем виде:

```
Treeptr  talloc(void)
{
   return (Treeptr) malloc(sizeof(Treenode));
}
```

Следует подчеркнуть, что объявление typedef не создает объявления нового типа, оно лишь сообщает новое имя уже существующему типу. Никакого нового смысла эти новые имена не несут, они объявляют переменные в точности с теми же свойствами, как если бы те были объявлены напрямую без переименования типа. Фактически typedef аналогичен #define с тем лишь отличием, что при интерпретации компилятором он

может справиться с такой текстовой подстановкой, которая не может быть обработана препроцессором. Например

```
typedef int (*PFI)(char *, char *);
```

создает тип PFI - "указатель на функцию (двух аргументов типа char *), возвращающую int", который, например, в программе сортировки, описанной в главе 5, можно использовать в таком контексте:

```
PFI strcmp, numcmp;
```

Помимо просто эстетических соображений, для применения typedef существуют две важные причины. Первая - параметризация программы, связанная с проблемой переносимости. Если с помощью typedef объявить типы данных, которые, возможно, являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения typedef. Одна из распространенных ситуаций - использование typedef-имен для варьирования целыми величинами. Для каждой конкретной машины это предполагает соответствующие установки short, int или long, которые делаются аналогично установкам стандартных типов, например size_t и ptrdiff_t.

Вторая причина, побуждающая к применению typedef,— желание сделать более ясным текст программы. Тип, названный Treeptr (от английских слов tree - дерево и pointer - указатель), более понятен, чем тот же тип, записанный как указатель на некоторую сложную структуру.

6.8. Объединения

Объединение - это переменная, которая может содержать (в разные моменты времени) объекты различных типов и размеров. Все требования относительно размеров и выравнивания выполняет компилятор. Объединения позволяют хранить разнородные данные в одной и той же области памяти без включения в программу машинно-зависимой информации. Эти средства аналогичны вариантным записям в Паскале.

Примером использования объединений мог бы послужить сам компилятор, заведующий таблицей символов, если предположить, что константа может иметь тип int, float или являться указателем на символ и иметь тип char *. Значение каждой конкретной константы должно храниться в переменной соответствующего этой константе типа. Работать с таблицей символов всегда удобнее, если значения занимают одинаковую по объему память и запоминаются в одном и том же месте независимо от своего типа. Цель введения в программу объединения - иметь переменную, которая бы на законных основаниях хранила в себе значения нескольких

типов. Синтаксис объединений аналогичен синтаксису структур. Приведем пример объединения.

```
union u_tag {
   int ival;
   float fval;
   char *sval;
} u;
```

Переменная u будет достаточно большой, чтобы в ней поместилась любая переменная из указанных трех типов; точный ее размер зависит от реализации. Значение одного из этих трех типов может быть присвоено переменной u и далее использовано в выражениях, если это правомерно, т. е. если тип взятого ею значения совпадает с типом последнего присвоенного ей значения. Выполнение этого требования в каждый текущий момент - целиком на совести программиста. В том случае, если нечто запомнено как значение одного типа, а извлекается как значение другого типа, результат зависит от реализации.

Синтаксис доступа к элементам объединения следующий:

имя-объединения . *элемент*

или

указатель-на-объединение -> *элемент*

т. е. в точности такой, как в структурах. Если для хранения типа текущего значения u использовать, скажем, переменную utype, то можно написать такой фрагмент программы:

```
if (utype == INT)
   printf("%d\n", u.ival);
else if (utype == FLOAT)
   printf("%f\n", u.fval);
else if (utype == STRING)
   printf("%s\n", u.sval);
else
   printf ("неверный тип %d в utype\n", utype);
```

Объединения могут входить в структуры и массивы, и наоборот. Запись доступа к элементу объединения, находящегося в структуре (как и структуры, находящейся в объединении), такая же, как и для вложенных структур. Например, в массиве структур

```
struct {
   char *name;
   int flags;
```

```
int utype;
union {
    int ival;
    float fval;
    char *sval;
} u;
} symtab[NSYM];
```

к ival обращаются следующим образом:

```
symtab[i].u.ival
```

а к первому символу строки sval можно обратиться любым из следующих двух способов:

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

Фактически объединение - это структура, все элементы которой имеют нулевое смещение относительно ее базового адреса и размер которой позволяет поместиться в ней самому большому ее элементу, а выравнивание этой структуры удовлетворяет всем типам объединения. Операции, применимые к структурам, годятся и для объединений, т. е. законны присваивание объединения и копирование его как единого целого, взятие адреса от объединения и доступ к отдельным его элементам.

Инициализировать объединение можно только значением, имеющим тип его первого элемента; таким образом, упомянутую выше переменную u можно инициализировать лишь значением типа int.

В главе 8 (на примере программы, заведующей выделением памяти) мы покажем, как, применяя объединение, можно добиться, чтобы расположение переменной было выровнено по соответствующей границе в памяти.

6.9. Битовые поля

При дефиците памяти может возникнуть необходимость запаковать несколько объектов в одно слово машины. Одна из обычных ситуаций, встречающаяся в задачах обработки таблиц символов для компиляторов, - это объединение групп однобитовых флажков. Форматы некоторых данных могут от нас вообще не зависеть и диктоваться, например, интерфейсами с аппаратурой внешних устройств; здесь также возникает потребность адресоваться к частям слова.

Вообразим себе фрагмент компилятора, который заведует таблицей символов. Каждый идентификатор программы имеет некоторую связанную

с ним информацию: например, представляет ли он собой ключевое слово и, если это переменная, к какому классу принадлежит: внешняя и / или статическая и т. д. Самый компактный способ кодирования такой информации - расположить однобитовые флажки в одном слове типа char или int.

Один из распространенных приемов работы с битами основан на определении набора "масок", соответствующих позициям этих битов, как, например, в

```
#define KEYWORD 01      /* ключевое слово */
#define EXTERNAL 02     /* внешний */
#define STATIC 04       /* статический */
```

или в

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Числа должны быть степенями двойки. Тогда доступ к битам становится делом "побитовых операций", описанных в главе 2 (сдвиг, маскирование, взятие дополнения).

Некоторые виды записи выражений встречаются довольно часто. Так,

```
flags ¦= EXTERNAL ! STATIC:
```

устанавливает 1 в соответствующих битах переменной flags,

```
flags &= ~(EXTERNAL i STATIC);
```

обнуляет их, а

```
if ((flags & (EXTERNAL i STATIC)) == 0) ...
```

оценивает условие как истинное, если оба бита нулевые.

Хотя научиться писать такого рода выражения не составляет труда, вместо побитовых логических операций можно пользоваться предоставляемым Си другим способом прямого определения и доступа к полям внутри слова. *Битовое поле* (или для краткости просто *поле)* - это некоторое множество битов, лежащих рядом внутри одной, зависящей от реализации единицы памяти, которую мы будем называть "словом". Синтаксис определения полей и доступа к ним базируется на синтаксисе структур. Например, строкиtfdefine, фигурировавшие выше при задании таблицы символов, можно заменить на определение трех полей:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
> flags;
```

Эта запись определяет переменную flags, которая содержит три однобитовых поля. Число, следующее за двоеточием, задает ширину поля. Поля объявлены как unsigned int, чтобы они воспринимались как беззнаковые величины.

На отдельные поля ссылаются так же, как и на элементы обычных структур: flags. is_keyword, flags.is_extern и т. д. Поля "ведут себя" как малые целые и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры можно написать более естественно:

```
flags.is_extern = flags.is_static = 1;
```

устанавливает 1 в соответствующие биты;

```
flags.is_extern = flags.is_static = 0;
```

их обнуляет, а

```
if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

проверяет их.

Почти все технические детали, касающиеся полей, в частности, возможность поля перейти границу слова, зависят от реализации. Поля могут не иметь имени; с помощью безымянного поля (задаваемого только двоеточием и шириной) организуется пропуск нужного количества разрядов. Особая ширина, равная нулю, используется, когда требуется выйти на границу следующего слова.

На одних машинах поля размещаются слева направо, на других - справа налево. Это значит, что при всей полезности работы с ними, если формат данных, с которыми мы имеем дело, дан нам свыше, то необходимо самым тщательным образом исследовать порядок расположения полей; программы, зависящие от такого рода вещей, не переносимы. Поля можно определять только с типом int, а для того чтобы обеспечить переносимость, надо явно указывать signed или unsigned. Они не могут быть массивами и не имеют адресов, и, следовательно, оператор & к ним не применим.

Глава 7

Ввод и вывод

Возможности для ввода и вывода не являются частью самого языка Си, поэтому мы подробно и не рассматривали их до сих пор. Между тем реальные программы взаимодействуют со своим окружением гораздо более сложным способом, чем те, которые были затронуты ранее. В этой главе мы опишем стандартную библиотеку, содержащую набор функций, обеспечивающих ввод-вывод, работу со строками, управление памятью, стандартные математические функции и разного рода сервисные Си-программы. Но особое внимание уделим вводу-выводу.

Библиотечные функции ввода-вывода точно определяются стандартом ANSI, так что они совместимы на любых системах, где поддерживается Си. Программы, которые в своем взаимодействии с системным окружением не выходят за рамки возможностей стандартной библиотеки, можно без изменений переносить с одной машины на другую.

Свойства библиотечных функций специфицированы в более чем дюжине заголовочных файлов; вам уже встречались некоторые из них, в том числе <stdio.h>, <string.h> и <ctype.h>. Мы не рассматриваем здесь всю библиотеку, так как нас больше интересует написание Си-программ, чем использование библиотечных функций. Стандартная библиотека подробно описана в приложении В.

7.1. Стандартный ввод-вывод

Как уже говорилось в главе 1, библиотечные функции реализуют простую модель текстового ввода-вывода. Текстовый поток состоит из последовательности строк; каждая строка заканчивается символом новой строки. Если система в чем-то не следует принятой модели, библиотека сделает так, чтобы казалось, что эта модель удовлетворяется полностью. Например, пара символов - возврат-каретки и перевод-строки - при вво-

де могла бы быть преобразована в один символ новой строки, а при выводе выполнялось бы обратное преобразование.

Простейший механизм ввода - это чтение одного символа из *стандартного ввода* (обычно с клавиатуры) функцией getchar:

```
int getchar(void)
```

В качестве результата каждого своего вызова функция getchar возвращает следующий символ ввода или, если обнаружен конец файла, EOF. Именованная константа EOF (аббревиатура от *end of file* - конец файла) определена в <stdio.h>. Обычно значение EOF равно -1, но, чтобы не зависеть от конкретного значения этой константы, обращаться к ней следует по имени (EOF).

Во многих системах клавиатуру можно заменить файлом, перенаправив ввод с помощью значка <. Так, если программа prog использует getchar, то командная строка

```
prog < infile
```

предпишет программе prog читать символы из infile, а не с клавиатуры. Переключение ввода делается так, что сама программа prog не замечает подмены; в частности строка "<infile" не будет включена в аргументы командной строки argv. Переключение ввода будет также незаметным, если ввод исходит от другой программы и передается конвейерным образом. В некоторых системах командная строка

```
otherprog ! prog
```

приведет к тому, что запустится две программы, otherprog и prog, и стандартный выход otherprog поступит на стандартный вход prog.

Функция

```
int putchar(int)
```

используется для вывода: putchar(c) отправляет символ с в *стандартный вывод,* под которым по умолчанию подразумевается экран. Функция putchar в качестве результата возвращает посланный символ или, в случае ошибки, EOF. То же и в отношении вывода: с помощью записи вида > *имя-файла* вывод можно перенаправить в файл. Например, если prog использует для вывода функцию putchar, то

```
prog > outfile
```

будет направлять стандартный вывод не на экран, а в outfile. А командная строка

```
prog ¦ anotherprog
```

соединит стандартный вывод программы prog со стандартным вводом программы anotherprog.

Вывод, осуществляемый функцией printf, также отправляется в стандартный выходной поток. Вызовы putchar и printf могут как угодно чередоваться, при этом вывод будет формироваться в той последовательности, в которой происходили вызовы этих функций.

Любой исходный Си-файл, использующий хотя бы одну функцию библиотеки ввода-вывода, должен содержать в себе строку

```
#include<stdio. h>
```

причем она должна быть расположена до первого обращения к вводу-выводу. Если имя заголовочного файла заключено в угловые скобки < и >, это значит, что поиск заголовочного файла ведется в стандартном месте (например в системе UNIX это обычно директорий /usr/include).

Многие программы читают только из одного входного потока и пишут только в один выходной поток. Для организации ввода-вывода таким программам вполне хватит функций getchar, putchar и printf, а для начального обучения уж точно достаточно ознакомления с этими функциями. В частности, перечисленных функций достаточно, когда требуется вывод одной программы соединить с вводом следующей. В качестве примера рассмотрим программу lower, переводящую свой ввод на нижний регистр:

```
#include<stdio.h>
#include<ctype.h>

  main() /* lower: переводит ввод на нижний регистр */
  {
  int c;

    while ((c = getchar()) != EOF)
      putchar(tolower(c));
    return 0;
  }
```

Функция tolower определена в <ctype. h>. Она переводит буквы верхнего регистра в буквы нижнего регистра, а остальные символы возвращает без изменений. Как мы уже упоминали, "функции" вроде getchar и putchar из библиотеки <stdio.h> и функция tolower из библиотеки <ctype. h> часто реализуются в виде макросов, чтобы исключить накладные расходы от вызова функции на каждый отдельный символ. В параграфе 8.5 мы покажем, как это делается. Независимо от того, как на той или иной машине реализованы функции библиотеки <ctype. h>, использующие их программы могут ничего не знать о кодировке символов.

Упражнение 7.1. Напишите программу, осуществляющую перевод ввода с верхнего регистра на нижний или с нижнего на верхний в зависимости от имени, по которому она вызывается и текст которого находится в `arg[0]`.

7.2. Форматный вывод (printf)

Функция `printf` переводит внутренние значения в текст.

`int printf(char *format,` arg_1, arg_2, `...)`

В предыдущих главах мы использовали `printf` неформально. Здесь мы покажем наиболее типичные случаи применения этой функции; полное ее описание дано в приложении B.

Функция `printf` преобразует, форматирует и печатает свои аргументы в стандартном выводе под управлением формата. Возвращает она количество напечатанных символов.

Форматная строка содержит два вида объектов: обычные символы, которые впрямую копируются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента `printf`. Любая спецификация преобразования начинается знаком % и заканчивается символом-спецификатором. Между % и символом-спецификатором могут быть расположены (в указанном ниже порядке) следующие элементы:

- Знак минус, предписывающий выравнивать преобразованный аргумент по левому краю поля.
- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет занимать поле по крайней мере указанной ширины. При необходимости лишние позиции слева (или справа при левостороннем расположении) будут заполнены пробелами.
- Точка, отделяющая ширину поля от величины, устанавливающей точность.
- Число (точность), специфицирующее максимальное количество печатаемых символов в строке, или количество цифр после десятичной точки - для чисел с плавающей запятой, или минимальное количество цифр - для целого.
- Буква h, если печатаемое целое должно рассматриваться как `short`, или l (латинская буква ell), если целое должно рассматриваться как `long`.

Символы-спецификаторы перечислены в в таблице 7.1. Если за % не помещен символ-спецификатор, поведение функции `printf` будет не определено.

Ширину и точность можно специфицировать с помощью *; значение ширины (или точности) в этом случае берется из следующего аргумента

Таблица 7.1. **Основные преобразования** printf

Символ	Тип аргумента; вид печати
d, i	int; десятичное целое
o	int; беззнаковое восьмеричное (*octal*) целое (без нуля слева)
x, X	unsigned int; беззнаковое шестнадцатеричное целое (без 0x или 0X слева), для 10...15 используются abcdef или ABCDEF
u	int; беззнаковое десятичное целое
c	int; одиночный символ
s	char *; печатает символы, расположенные до знака \0, или в количестве, заданном точностью
f	double; [-] *m.dddddd*, где количество цифр *d* задается точностью (по умолчанию равно 6)
e, E	double; [-] *m.dddddd*e±*xx* или [-] *m.dddddd*E±*xx*, где количество цифр *d* задается точностью (по умолчанию равно 6)
g, G	double; использует %e или %E, если порядок меньше, чем -4, или больше или равен точности; в противном случае использует %f. Завершающие нули и завершающая десятичная точка не печатаются
p	void *; указатель (представление зависит от реализации)
%	Аргумент не преобразуется; печатается знак %

(который должен быть типа int). Например, чтобы напечатать не более max символов из строки s, годится следующая запись:

```
printf("%.*s", max, s);
```

Большая часть форматных преобразований была продемонстрирована в предыдущих главах. Исключение составляет задание точности для строк. Далее приводится перечень спецификаций и показывается их влияние на печать строки "hello, world", состоящей из 12 символов. Поле специально обрамлено двоеточиями, чтобы была видна его протяженность.

```
:%s:        :hello, world:
:%10s:      :hello, world:
:%.10s:     :hello, wor:
:%-10s:     :hello, world:
:%.15s:     :hello, world:
:%-15s:     :hello, world   :
:%15.10s:   :     hello, wor:
:%-15.10s:  :hello, wor     :
```

Предостережение: функция printf использует свой первый аргумент, чтобы определить, сколько еще ожидается аргументов и какого они будут типа. Вы не получите правильного результата, если аргументов будет не хватать или они будут принадлежать не тому типу. Вы должны также понимать разницу в следующих двух обращениях:

```
printf(s);          /* НЕВЕРНО, если в s есть %, */
printf("%s", s);  /* ВЕРНО всегда */
```

Функция sprintf выполняет те же преобразования, что и printf, но вывод запоминает в строке

int sprintf(char *string, char *format, arg_1, arg_2, ...)

Эта функция форматирует arg_1, arg_2 и т. д. в соответствии с информацией, заданной аргументом format, как мы описывали ранее, но результат помещает не в стандартный вывод, а в string. Заметим, что строка string должна быть достаточно большой, чтобы в ней поместился результат.

Упражнение 7.2. Напишите программу, которая будет печатать разумным способом любой ввод. Как минимум она должна уметь печатать неграфические символы в восьмеричном или шестнадцатеричном виде (в форме, принятой на вашей машине), обрывая длинные текстовые строки.

7.3. Списки аргументов переменной длины

Этот параграф содержит реализацию минимальной версии printf. Приводится она для того, чтобы показать, как надо писать функции со списками аргументов переменной длины, причем такие, которые были бы переносимы. Поскольку нас главным образом интересует обработка аргументов, функцию minprintf напишем таким образом, что она в основном будет работать с задающей формат строкой и аргументами; что же касается форматных преобразований, то они будут осуществляться с помощью стандартного printf.

Объявление стандартной функции printf выглядит так:

```
int printf(char *fmt, ...)
```

Многоточие в объявлении означает, что число и типы аргументов могут изменяться. Знак многоточие может стоять только в конце списка аргументов. Наша функция minprintf объявляется как

```
void minprintf(char *fmt, ...)
```

поскольку она не будет выдавать число символов, как это делает printf.

Вся сложность в том, каким образом minprintf будет продвигаться вдоль списка аргументов, - ведь у этого списка нет даже имени. Стандартный заголовочный файл <stdarg.h> содержит набор макроопределений, которые устанавливают, как шагать по списку аргументов. Наполнение этого заголовочного файла может изменяться от машины к машине, но представленный им интерфейс везде одинаков.

Тип va_list служит для описания переменной, которая будет по очереди указывать на каждый из аргументов; в minprintf эта переменная имеет имя ap (от *"argument pointer"* - указатель на аргумент). Макрос va_start инициализирует переменную ap, чтобы она указывала на первый безымянный аргумент. К va_start нужно обратиться до первого использования ap. Среди аргументов по крайней мере один должен быть именованным; от последнего именованного аргумента этот макрос "отталкивается" при начальной установке.

Макрос va_arg на каждом своем вызове выдает очередной аргумент, а ap передвигает на следующий; по имени типа он определяет тип возвращаемого значения и размер шага для выхода на следующий аргумент. Наконец, макрос va_end делает очистку всего, что необходимо. К va_end следует обратиться перед самым выходом из функции.

Перечисленные средства образуют основу нашей упрощенной версии printf.

```
#include <stdarg.h>

/* minprintf: минимальный printf с переменным числом аргумент */
void minprintf(char *fmt, ...)
{
  va_list ap;   /* указывает на очередной безымянный аргумент */
  char *p, *sval;
  int ival;
  double dval;

  va_start(ap, fmt); /* устанавливает ap на 1-й безымянный аргумент */
  for (p = fmt; *p; p++) {
    if (*p != '%') {
      putchar(*p);
      continue;
    }
    switch (*++p) {
    case 'd':
      ival = va_arg(ap, int);
      printf ("%d", ival);
```

```
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap); /* очистка, когда все сделано */
}
```

Упражнение 7.3. Дополните minprintf другими возможностями printf.

7.4. Форматный ввод (scanf)

Функция scanf, обеспечивающая ввод, является аналогом printf; она выполняет многие из упоминавшихся преобразований, но в противоположном направлении. Ее объявление имеет следующий вид:

```
int scanf(char *format, ...)
```

Функция scanf читает символы из стандартного входного потока, интерпретирует их согласно спецификациям строки format и рассылает результаты в свои остальные аргументы. Аргумент-формат мы опишем позже; другие аргументы, *каждый из которых должен быть указателем,* определяют, где будут запоминаться должным образом преобразованные данные. Как и для printf, в этом параграфе дается сводка наиболее полезных, но отнюдь не всех возможностей данной функции.

Функция scanf прекращает работу, когда оказывается, что исчерпался формат или вводимая величина не соответствует управляющей спецификации. В качестве результата scanf возвращает количество успешно введенных элементов данных. По исчерпании файла она выдает EOF. Существенно то, что значение EOF не равно нулю, поскольку нуль scanf выдает, когда вводимый символ не соответствует первой спецификации форматной строки. Каждое очередное обращение к scanf продолжает ввод с символа, следующего сразу за последним обработанным.

Существует также функция sscanf, которая читает из строки (а не из стандартного ввода).

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Функция sscanf просматривает строку string согласно формату format и рассылает полученные значения в arg_1, arg_2 и т. д. Последние должны быть указателями.

Формат обычно содержит спецификации, которые используются для управления преобразованиями ввода. В него могут входить следующие элементы:

- Пробелы или табуляции, которые игнорируются.
- Обычные символы (исключая %), которые, как ожидается, совпадут с очередными символами, отличными от символов-разделителей входного потока.
- Спецификации преобразования, каждая из которых начинается со знака % и завершается символом-спецификатором типа преобразования. В промежутке между этими двумя символами в любой спецификации могут располагаться, причем в том порядке, как они здесь указаны: знак * (признак подавления присваивания); число, определяющее ширину поля; буква h, l или L, указывающая на размер получаемого значения; и символ преобразования (o, d, x).

Спецификация преобразования управляет преобразованием следующего вводимого поля. Обычно результат помещается в переменную, на которую указывает соответствующий аргумент. Однако если в спецификации преобразования присутствует *, то поле ввода пропускается и никакое присваивание не выполняется. Поле ввода определяется как строка без символов-разделителей; оно простирается до следующего символа-разделителя или же ограничено шириной поля, если она задана. Поскольку символ новой строки относится к символам-разделителям, то scanf при чтении будет переходить с одной строки на другую. (Символами-разделителями являются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и перевода страницы.)

Символ-спецификатор указывает, каким образом следует интерпретировать очередное поле ввода. Соответствующий аргумент должен быть указателем, как того требует механизм передачи параметров по значению, принятый в Си. Символы-спецификаторы приведены в таблице 7.2.

Перед символами-спецификаторами d, i, o, u и x может стоять буква h, указывающая на то, что соответствующий аргумент должен иметь тип short * (а не int *), или l (латинская ell), указывающая на тип long *. Аналогично, перед символами-спецификаторами e, f и g может стоять буква l, указывающая, что тип аргумента - double * (а не float *).

Таблица 7.2. Основные преобразования scanf

Символ	Вводимые данные; тип аргумента
d	десятичное целое; int *
i	целое; int *. Целое может быть восьмеричным (с 0 слева) или шестнадцатеричным (с 0x или 0X слева)
o	восьмеричное целое (с нулем слева или без него); int *
u	беззнаковое десятичное целое; unsigned int *
x	шестнадцатеричное целое (с 0x или 0X слева или без них); int *
c	символы; char *. Следующие символы ввода (по умолчанию один) размещаются в указанном месте. Обычный пропуск символов-разделителей подавляется; чтобы прочесть очередной символ, отличный от символа-разделителя, используйте %1s
s	строка символов (без обрамляющих кавычек); char *, указывающая на массив символов, достаточный для строки и завершающего символа '\0', который будет добавлен
e, f, g	число с плавающей точкой, возможно, со знаком; обязательно присутствие либо десятичной точки, либо экспоненциальной части, а возможно, и обеих вместе; float *
%	сам знак %, никакое присваивание не выполняется

Чтобы построить первый пример, обратимся к программе калькулятора из главы 4, в которой организуем ввод с помощью функции scanf:

```
#include <stdio.h>

main()  /* программа-калькулятор */
{
  double sum, v;

  sum = 0;
  while (scanf ("%lf", &v) == 1)
     printf("\t%.2f\n", sum += v);
  return 0;
}
```

Предположим, что нам нужно прочитать строки ввода, содержащие данные вида

```
25 дек 1988
```

Обращение к scanf выглядит следующим образом:

```
int day, year;        /* день, год */
char monthname[20];   /* название месяца */

scanf ("%d %s %d", &day, monthname, &year);
```

Знак & перед monthname не нужен, так как имя массива есть указатель.

В строке формата могут присутствовать символы, не участвующие ни в одной из спецификаций; это значит, что эти символы должны появиться на вводе. Так, мы могли бы читать даты вида mm/dd/yy с помощью следующего обращения к scanf:

```
int day, month, year;     /* день, месяц, год */
scanf("%d/%d/%d", &day, &month, &year);
```

В своем формате функция scanf игнорирует пробелы и табуляции. Кроме того, при поиске следующей порции ввода она пропускает во входном потоке все символы-разделители (пробелы, табуляции, новые строки и т. д.). Воспринимать входной поток, не имеющий фиксированного формата, часто оказывается удобнее, если вводить всю строку целиком и для каждого отдельного случая подбирать подходящий вариант sscanf. Предположим, например, что нам нужно читать строки с датами, записанными в любой из приведенных выше форм. Тогда мы могли бы написать:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("верно: %s\n", line);     /* в виде 25 дек 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("верно: %s\n", line);     /* в виде mm/dd/yy */
    else
        printf("неверно: %s\n", line);   /* неверная форма даты */
}
```

Обращения к scanf могут перемежаться с вызовами других функций ввода. Любая функция ввода, вызванная после scanf, продолжит чтение с первого еще непрочитанного символа.

В завершение еще раз напомним, что аргументы функций scanf и sscanf *должны* быть указателями.

Одна из самых распространенных ошибок состоит в том, что вместо того, чтобы написать

```
scanf("%d", &n);
```

пишут

```
scanf("%d", n);
```

Компилятор о подобной ошибке ничего не сообщает.

Упражнение 7.4. Напишите свою версию scanf по аналогии с minprintf из предыдущего параграфа.

Упражнение 7.5. Перепишите основанную на постфиксной записи программу калькулятора из главы 4 таким образом, чтобы для ввода и преобразования чисел она использовала scanf и/или sscanf.

7.5. Доступ к файлам

Во всех предыдущих примерах мы имели дело со стандартным вводом и стандартным выводом, которые для программы автоматически предопределены операционной системой конкретной машины.

Следующий шаг - научиться писать программы, которые имели бы доступ к файлам, заранее не подсоединенным к программам. Одна из программ, в которой возникает такая необходимость, - это программа cat, объединяющая несколько именованных файлов и направляющая результат в стандартный вывод. Функция cat часто применяется для выдачи файлов на экран, а также как универсальный "коллектор" файловой информации для тех программ, которые не имеют возможности обратиться к файлу по имени. Например, команда

```
cat x.c y.c
```

направит в стандартный вывод содержимое файлов x. c и y. c (и ничего более).

Возникает вопрос: что надо сделать, чтобы именованные файлы можно было читать; иначе говоря, как связать внешние имена, придуманные пользователем, с инструкциями чтения данных?

На этот счет имеются простые правила. Для того чтобы можно было читать из файла или писать в файл, он должен быть предварительно *открыт с* помощью библиотечной функции fopen. Функция fopen получает внешнее имя типа x. c или y. c, после чего осуществляет некоторые организационные действия и "переговоры" с операционной системой (технические детали которых здесь не рассматриваются) и возвращает указатель, используемый в дальнейшем для доступа к файлу.

Этот указатель, называемый *указателем файла,* ссылается на структуру, содержащую информацию о файле (адрес буфера, положение текущего символа в буфере, открыт файл на чтение или на запись, были ли ошибки при работе с файлом и не встретился ли конец файла). Пользователю не нужно знать подробности, поскольку определения, полученные из <stdio.h>, включают описание такой структуры, называемой FILE.

Единственное, что требуется для определения указателя файла, - это задать описания такого, например, вида:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Это говорит, что fp есть указатель на FILE, a fopen возвращает указатель на FILE. Заметим, что FILE — это имя типа, наподобие int, а не тег структуры. Оно определено с помощью typedef. (Детали того, как можно реализовать fopen в системе UNIX, приводятся в параграфе 8.5.)

Обращение к fopen в программе может выглядеть следующим образом:

```
fp = fopen(name, mode);
```

Первый аргумент - строка, содержащая имя файла. Второй аргумент несет информацию о *режиме*. Это тоже строка: в ней указывается, каким образом пользователь намерен применять файл. Возможны следующие режимы: чтение (*read* – "r"), запись (*write* - "w") и добавление (*append* – "a"), т. е. запись информации в конец уже существующего файла. В некоторых системах различаются текстовые и бинарные файлы; в случае последних в строку режима необходимо добавить букву "b" (*binary* - бинарный).

Тот факт, что некий файл, которого раньше не было, открывается на запись или добавление, означает, что он создается (если такая процедура физически возможна). Открытие уже существующего файла на запись приводит к выбрасыванию его старого содержимого, в то время как при открытии файла на добавление его старое содержимое сохраняется. Попытка читать несуществующий файл является ошибкой. Могут иметь место и другие ошибки; например, ошибкой считается попытка чтения файла, который по статусу запрещено читать. При наличии любой ошибки fopen возвращает NULL. (Возможна более точная идентификация ошибки; детальная информация по этому поводу приводится в конце параграфа 1 приложения В.)

Следующее, что нам необходимо знать, - это как читать из файла или писать в файл, коль скоро он открыт. Существует несколько способов сделать это, из которых самый простой состоит в том, чтобы воспользоваться функциями getc и putc. Функция getc возвращает следующий символ из файла; ей необходимо сообщить указатель файла, чтобы она знала откуда брать символ.

```
int getc(FILE *fp)
```

Функция getc возвращает следующий символ из потока, на который указывает *fp; в случае исчерпания файла или ошибки она возвращает EOF.

Функция putc пишет символ c в файл fp

```
int putc(int c, FILE *fp)
```

и возвращает записанный символ или EOF в случае ошибки. Аналогично getchar и putchar, реализация getc и putc может быть выполнена в виде макросов, а не функций.

При запуске Си-программы операционная система всегда открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок; соответствующие им указатели называются stdin, stdout и stderr; они описаны в <stdio.h>. Обычно stdin соотнесен с клавиатурой, а stdout и stderr - с экраном. Однако stdin и stdout можно связать с файлами или, используя конвейерный механизм, соединить напрямую с другими программами, как это описывалось в параграфе 7.1.

С помощью getc, putc, stdin и stdout функции getchar и putchar теперь можно определить следующим образом:

```
fldefine getchar()   getc(stdin)
tfdefine putchar(c)  putc((c), stdout)
```

Форматный ввод-вывод файлов можно построить на функциях fscanf и fprintf. Они идентичны scanf и printf с той лишь разницей, что первым их аргументом является указатель на файл, для которого осуществляется ввод-вывод, формат же указывается вторым аргументом.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Вот теперь мы располагаем теми сведениями, которые достаточны для написания программы cat, предназначенной для конкатенации (последовательного соединения) файлов. Предлагаемая версия функции cat, как оказалось, удобна для многих программ. Если в командной строке присутствуют аргументы, они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается стандартный ввод.

```
ttinclude <stdio.h>

/* cat: конкатенация файлов, версия 1 */
main(int argc, char *argv[])
{
  FILE *fp;
  void filecopy(FILE *, FILE *);

  if (argc == 1)  /* нет аргументов; копируется стандартный ввод */
    filecopy(stdin, stdout);
  else
    while (--argc > 0)
```

```
        if ((fp = fopen(*++argv, "r")) == NULL) {
            printf("cat: не могу открыть файл %s\n", *argv);
            return 1;
        } else {
            filecopy(fp, stdout);
            fclose(fp);
        }
    return 0;
}

/* filecopy: копирует файл ifp в файл ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Файловые указатели stdin и stdout представляют собой объекты типа FILE*. Это константы, а *не* переменные, следовательно, им нельзя ничего присваивать.

Функция

```
int fclose(FILE *fp)
```

- обратная по отношению к fopen; она разрывает связь между файловым указателем и внешним именем (которая раньше была установлена с помощью fopen), освобождая тем самым этот указатель для других файлов. Так как в большинстве операционных систем количество одновременно открытых одной программой файлов ограничено, то файловые указатели, если они больше не нужны, лучше освобождать, как это и делается в программе cat. Есть еще одна причина применить fclose к файлу вывода, – это необходимость "опорожнить" буфер, в котором putc накопила предназначенные для вывода данные. При нормальном завершении работы программы для каждого открытого файла fclose вызывается автоматически. (Вы можете закрыть stdin и stdout, если они вам не нужны. Воспользовавшись библиотечной функцией freopen, их можно восстановить.)

7.6. Управление ошибками (stderr и exit)

Обработку ошибок в cat нельзя признать идеальной. Беда в том, что если файл по какой-либо причине недоступен, сообщение об этом мы получим по окончании конкатенируемого вывода. Это нас устроило бы, если

бы вывод отправлялся только на экран, а не в файл или по конвейеру другой программе.

Чтобы лучше справиться с этой проблемой, программе помимо стандартного вывода stdout придается еще один выходной поток, называемый stderr. Вывод в stderr обычно отправляется на экран, даже если вывод stdout перенаправлен в другое место.

Перепишем cat так, чтобы сообщения об ошибках отправлялись в stderr.

```
#include <stdio.h>

/* cat: конкатенация файлов, версия 2 */
main(int argc, char *argv[])
{
   FILE *fp;
   void filecopy(FILE *, FILE *);
   char *prog = argv[0];  /* имя программы */

   if (argc == 1)  /* нет аргументов; копируется станд. ввод */
      filecopy(stdin,  stdout);
   else
      while (--argc > 0)
         if ((fp = fopen(*++argv, "r")) == NULL) {
            fprintf(stderr, "%s: не могу открыть файл %s\n",
            prog, *argv);
            exit(1);
         } else {
            filecopy(fp,  stdout);
            fclose(fp);
         }
   if (ferror(stdout)) {
      fprintf(stderr, "%s: ошибка записи в stdout\n", prog);
      exit(2);
   }
   exit(0);
}
```

Программа сигнализирует об ошибках двумя способами. Первый - сообщение об ошибке с помощью fprintf посылается в stderr с тем, чтобы оно попало на экран, а не оказалось на конвейере или в другом файле вывода. Имя программы, хранящееся в argv[0], мы включили в сообщение, чтобы в слу/чаях, когда данная программа работает совместно с другими, был ясен источник ошибки.

Второй способ указать на ошибку — обратиться к библиотечной функции exit, завершающей работу программы. Аргумент функции exit доступен некоторому процессу, вызвавшему данный процесс. А следовательно, успешное или ошибочное завершение программы можно проконтролировать с помощью некоей программы, которая рассматривает эту программу в качестве подчиненного процесса. По общей договоренности возврат нуля сигнализирует о том, что работа прошла нормально, в то время как ненулевые значения обычно говорят об ошибках. Чтобы опорожнить буфера, накопившие информацию для всех открытых файлов вывода, функция exit вызывает fclose.

Инструкция return *выр* главной программы main эквивалентна обращению к функции exit(*выр*). Последний вариант (с помощью exit) имеет то преимущество, что он пригоден для выхода и из других функций, и, кроме того, слово exit легко обнаружить с помощью программы контекстного поиска, похожей на ту, которую мы рассматривали в главе 5.

Функция ferror выдает ненулевое значение, если в файле fp была обнаружена ошибка.

```
int ferror(FILE *fp)
```

Хотя при выводе редко возникают ошибки, все же они встречаются (например, оказался переполненным диск); поэтому в программах широкого пользования они должны тщательно контролироваться.

Функция feof(FILE *) аналогична функции ferror; она возвращает ненулевое значение, если встретился конец указанного в аргументе файла.

```
int feof(FILE *fp)
```

В наших небольших иллюстративных программах мы не заботились о выдаче статуса выхода, т. е. выдаче некоторого числа, характеризующего состояние программы в момент завершения: работа закончилась нормально или прервана из-за ошибки? Если работа прервана в результате ошибки, то какой? Любая серьезная программа должна выдавать статус выхода.

7.7. Ввод-вывод строк

В стандартной библиотеке имеется программа ввода fgets, аналогичная программе getline, которой мы пользовались в предыдущих главах.

```
char *fgets(char *line, int maxline, FILE *fp)
```

Функция fgets читает следующую строку ввода (включая и символ новой строки) из файла fp в массив символов line, причем она может прочитать не более MAXLINE-1 символов. Переписанная строка дополняется

символом '\0'. Обычно fgets возвращает line, а по исчерпании файла или в случае ошибки - NULL. (Наша getline возвращала длину строки, которой мы потом пользовались, и нуль в случае конца файла.)

Функция вывода fputs пишет строку (которая может и не заканчиваться символом новой строки) в файл.

```
int fputs(char *line, FILE *fp)
```

Эта функция возвращает EOF, если возникла ошибка, и неотрицательное значение в противном случае.

Библиотечные функции gets и puts подобны функциям fgets и fputs. Отличаются они тем, что оперируют только стандартными файлами stdin и stdout, и кроме того, gets выбрасывает последний символ '\n', а puts его добавляет.

Чтобы показать, что ничего особенного в функциях вроде fgets и fputs нет, мы приводим их здесь в том виде, в каком они существуют в стандартной библиотеке на нашей системе.

```
/* fgets: получает не более n символов из iop */
char *fgets(char *s, int n, FILE *iop)
{
  register int c;
  register char *cs;

  cs = s;
  while (--n > 0 && (c = getc(iop)) != EOF)
     if ((*cs++ = c) == '\n' )
        break;
  *cs = '\0';
  return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: посылает строку s в файл iop */
int fputs(char *s, FILE *iop)
{
  int c;

  while (c = *s++)
     putc(c, iop);
  return ferror(iop) ? EOF : 0;
}
```

Стандарт определяет, что функция ferror возвращает в случае ошибки ненулевое значение; fputs в случае ошибки возвращает EOF, в противном случае - неотрицательное значение.

С помощью fgets легко реализовать нашу функцию getline:

```
/* getline: читает строку, возвращает ее длину */
int getline(char *line, int max)
{
   if (fgets(line, max, stdin) == NULL)
      return 0;
   else
      return strlen(line);
}
```

Упражнение 7.6. Напишите программу, сравнивающую два файла и печатающую первую строку, в которой они различаются. •

Упражнение 7.7. Модифицируйте программу поиска по образцу из главы 5 таким образом, чтобы она брала текст из множества именованных файлов, а если имен файлов в аргументах нет, то из стандартного ввода. Будет ли печататься имя файла, в котором найдена подходящая строка?

Упражнение 7.8. Напишите программу, печатающую несколько файлов. Каждый файл должен начинаться с новой страницы, предваряться заголовком и иметь свою нумерацию страниц.

7.8. Другие библиотечные функции

В стандартной библиотеке представлен широкий спектр различных функций. Настоящий параграф содержит краткий обзор наиболее полезных из них. Более подробно эти и другие функции описаны в приложении В.

7.8.1. Операции со строками

Мы уже упоминали функции strlen, strcpy, strcat и strcmp, описание которых даны в <string.h>. Далее, до конца пункта, предполагается, что s и t имеют тип char *, с и n - тип int.

strcat(s,t)	- приписывает t в конец s.
strncat(s,t,n)	- приписывает n символов из t в конец s.
strcmp(s,t)	- возвращает отрицательное число, нуль или положительное число для s < t, s == t или s > t соответственно.
strncmp(s,t,n)	- делает то же, что и strcmp, но количество сравниваемых символов не может превышать n.
strcpy(s,t)	- копирует t в s.

strncpy(s, t, n) - копирует не более n символов из t в s.
strlen(s) - возвращает длину s.
strchr(s, c) - возвращает указатель на первое появление символа с в s или, если с нет в s, NULL.
strrchr(s, c) - возвращает указатель на последнее появление символа с в s или, если с нет в s, NULL.

7.8.2. Анализ класса символов и преобразование символов

Несколько функций из библиотеки <ctype.h> выполняют проверки и преобразование символов. Далее, до конца пункта, переменная с - это переменная типа int, которая может быть представлена значением unsigned char или EOF. Все эти функции возвращают значения типа int.

isalpha(c) - не нуль, если с - буква; 0 в противном случае.
isupper(c) - не нуль, если с - буква верхнего регистра; 0 в противном случае.
islower(c) - не нуль, если с - буква нижнего регистра; 0 в противном случае.
isdigit(c) - не нуль, если с - цифра; 0 в противном случае.
isalnum(c) - не нуль, если или isalpha(c), или isdigit(c) истинны; 0 в противном случае.
isspace(c) - не нуль, если с - символ пробела, табуляции, новой строки, возврата каретки, перевода страницы, вертикальной табуляции.
toupper(c) - возвращает с, приведенную к верхнему регистру.
tolower(c) - возвращает с, приведенную к нижнему регистру.

7.8.3. Функция ungetc

В стандартной библиотеке содержится более ограниченная версия функции ungetch по сравнению с той, которую мы написали в главе 4. Называется она ungetc. Эта функция, имеющая прототип

```
int ungetc(int c, FILE *fp)
```

отправляет символ с назад в файл fp и возвращает с, а в случае ошибки EOF. Для каждого файла гарантирован возврат не более одного символа. Функцию ungetc можно использовать совместно с любой из функций ввода вроде scanf, getc, getchar и т. д.

7.8.4. Исполнение команд операционной системы

Функция system(char *s) выполняет команду системы, содержащуюся в строке s, и затем возвращается к выполнению текущей программы.

Содержимое s, строго говоря, зависит от конкретной операционной системы. Рассмотрим простой пример: в системе UNIX инструкция

```
system("date");
```

вызовет программу date, которая направит дату и время в стандартный вывод. Функция возвращает зависящий от системы статус выполненной команды. В системе UNIX возвращаемый статус - это значение, переданное функцией exit.

7.8.5. Управление памятью

Функции malloc и calloc динамически запрашивают блоки свободной памяти. Функция malloc

```
void *malloc(size_t n)
```

возвращает указатель на n байт неинициализированной памяти или NULL, если запрос удовлетворить нельзя. Функция calloc

```
void *calloc(size_t n, size_t size)
```

возвращает указатель на область, достаточную для хранения массива из n объектов указанного размера (size), или NULL, если запрос не удается удовлетворить. Выделенная память обнуляется.

Указатель, возвращаемый функциями malloc и calloc, будет выдан с учетом выравнивания, выполненного согласно указанному типу объекта. Тем не менее к нему должна быть применена операция приведения к соответствующему типу[1], как это сделано в следующем фрагменте программы:

```
int *ip;
ip = (int *) calloc(n, sizeofF(int));
```

Функция free(p) освобождает область памяти, на которую указывает p, - указатель, первоначально полученный с помощью malloc или calloc. Никаких ограничений на порядок, в котором будет освобождаться память, нет, но считается ужасной ошибкой освобождение тех областей, которые не были получены с помощью calloc или malloc.

Нельзя также использовать те области памяти, которые уже освобождены. Следующий пример демонстрирует типичную ошибку в цикле, освобождающем элементы списка.

```
for (p = head; p != NULL; p = p->next)        /* НЕВЕРНО */
    free(p);
```

Правильным будет, если вы до освобождения сохраните то, что вам по-

[1] Как уже отмечалось (см. примеч. па с. 183), замечание о приведении типов значений, возвращаемых функциями malloc или calloc,— неверно. - *Примеч. акт.*

требуется, как в следующем цикле:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

В параграфе 8.7 мы рассмотрим реализацию программы управления памятью вроде malloc, позволяющую освобождать выделенные блоки памяти в любой последовательности.

7.8.6. Математические функции

В <math.h> описано более двадцати математических функций. Здесь же приведены наиболее употребительные. Каждая из них имеет один или два аргумента типа double и возвращает результат также типа double.

```
sin(x)        - синус x, x в радианах.
cos(x)        - косинус x, x в радианах.
atan2(y,x)    - арктангенс y/x, y и x в радианах.
exp(x)        - экспоненциальная функция e^x.
log(x)        - натуральный (по основанию e) логарифм x (x > 0).
log10(x)      - обычный (по основанию 10) логарифм x (x > 0).
pow(x,y)      - x^y.
sqrt(x)       - корень квадратный x (x > 0).
fabs(x)       - абсолютное значение x.
```

7.8.7. Генератор случайных чисел

Функция rand() вычисляет последовательность псевдослучайных целых в диапазоне от нуля до значения, заданного именованной константой RAND_MAX, которая определена в <stdlib.h>. Привести случайные числа к значениям с плавающей точкой, большим или равным 0 и меньшим 1, можно по формуле

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Если в вашей библиотеке уже есть функция для получения случайных чисел с плавающей точкой, вполне возможно, что ее статистические характеристики лучше указанной.)

Функция srand(unsigned) устанавливает семя для rand. Реализации rand и srand, предлагаемые стандартом и, следовательно, переносимые на различные машины, рассмотрены в параграфе 2.7.

Упражнение 7.9. Реализуя функции вроде isupper, можно экономить либо память, либо время. Напишите оба варианта функции.

Интерфейс с системой UNIX

Свои услуги операционная система UNIX предлагает в виде набора *системных вызовов,* которые фактически являются ее внутренними функциями и к которым можно обращаться из программ пользователя. В настоящей главе описано, как в Си-программах можно применять некоторые наиболее важные вызовы. Если вы работаете в системе UNIX, то эти сведения будут вам полезны непосредственно и позволят повысить эффективность работы или получить доступ к тем возможностям, которых нет в библиотеке. Даже если вы используете Си в другой операционной системе, изучение рассмотренных здесь примеров все равно приблизит вас к пониманию программирования на Си; аналогичные программы (отличающиеся лишь деталями) вы встретите практически в любой операционной системе. Так как библиотека Си-программ, утвержденная в качестве стандарта ANSI, в основном отражает возможности системы UNIX, предлагаемые программы помогут вам лучше понять и библиотеку.

Глава состоит из трех основных частей, описывающих: ввод-вывод, файловую систему и организацию управления памятью. В первых двух частях предполагается некоторое знакомство читателя с внешними характеристиками системы UNIX.

В главе 7 мы рассматривали единый для всех операционных систем интерфейс ввода-вывода. В любой конкретной системе программы стандартной библиотеки пишутся с использованием средств именно этой конкретной системы. В следующих нескольких параграфах мы опишем вызовы системы UNIX по вводу-выводу и покажем, как с их помощью можно реализовать некоторые разделы стандартной библиотеки.

8.1. Дескрипторы файлов

В системе UNIX любые операции ввода-вывода выполняются посредством чтения и записи файлов, поскольку все внешние устройства, вклю-

чая клавиатуру и экран, рассматриваются как объекты файловой системы. Это значит, что все связи между программой и внешними устройствами осуществляются в рамках единого однородного интерфейса.

В самом общем случае, прежде чем читать или писать, вы должны проинформировать систему о действиях, которые вы намереваетесь выполнять в отношении файла; эта процедура называется *открытием* файла. Если вы собираетесь писать в файл, то, возможно, его потребуется создать заново или очистить от хранимой информации. Система проверяет ваши права на эти действия (файл существует? вы имеете к нему доступ?) и, если все в порядке, возвращает программе небольшое неотрицательное целое, называемое *дескриптором файла.* Всякий раз, когда осуществляется ввод-вывод, идентификация файла выполняется по его дескриптору, а не по имени. (Дескриптор файла аналогичен файловому указателю, используемому в стандартной библиотеке, или хэндлу *(handle)* в MS-DOS.) Вся информация об открытом файле хранится и обрабатывается операционной системой; программа пользователя обращается к файлу только через его дескриптор.

Ввод с клавиатуры и вывод на экран применяются настолько часто, что для удобства работы с ними предусмотрены специальные соглашения. При запуске программы командный интерпретатор *(shell)* открывает три файла с дескрипторами 0, 1 и 2, которые называются соответственно стандартным вводом, стандартным выводом и стандартным файлом ошибок. Если программа читает из файла 0, а пишет в файлы 1 и 2 (здесь цифры - дескрипторы файлов), то она может осуществлять ввод и вывод, не заботясь об их открытии.

Пользователь программы имеет возможность перенаправить ввод-вывод в файл или из файла с помощью значков < и >, как, например, в

```
prog <infile >outfile
```

В этом случае командный интерпретатор заменит стандартные установки дескрипторов 0 и 1 на именованные файлы. Обычно дескриптор файла 2 остается подсоединенным к экрану, чтобы на него шли сообщения об ошибках. Сказанное верно и для ввода-вывода, связанного в конвейер. Во всех случаях замену файла осуществляет командный интерпретатор, а не программа. Программа, если она ссылается на файл 0 (в случае ввода) и файлы 1 и 2 (в случае вывода), не знает, ни откуда приходит ее ввод, ни куда отправляется ее вывод.

8.2. Нижний уровень ввода-вывода (read и write)

Ввод-вывод основан на системных вызовах read и write, к которым Си-программа обращается с помощью функций с именами read и write.

Для обеих первым аргументом является дескриптор файла. Во втором аргументе указывается массив символов вашей программы, куда посылаются или откуда берутся данные. Третий аргумент — это количество пересылаемых байтов.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Обе функции возвращают число переданных байтов. При чтении количество прочитанных байтов может оказаться меньше числа, указанного в третьем аргументе. Нуль означает конец файла, а -1 сигнализирует о какой-то ошибке. При записи функция возвращает количество записанных байтов, и если это число не совпадает с требуемым, следует считать, что запись не произошла.

За один вызов можно прочитать или записать любое число байтов. Обычно это число равно или 1, что означает посимвольную передачу "без буферизации", или чему-нибудь вроде 1024 или 4096, соответствующих размеру физического блока внешнего устройства. Эффективнее обмениваться большим числом байтов, поскольку при этом требуется меньше системных вызовов. Используя полученные сведения, мы можем написать простую программу, копирующую свой ввод на свой вывод и эквивалентную программе копирования файла, описанной в главе 1. С помощью этой программы можно копировать откуда угодно и куда угодно, поскольку всегда существует возможность перенаправить ввод-вывод на любой файл или устройство.

```
#include "syscalls.h"

main() /* копирование ввода на вывод */
{
  char buf[BUFSIZ];
  int n;

  while ((n = read(0, buf, BUFSIZ)) > 0)
    write(i, buf, n);
  return 0;
}
```

Прототипы функций, обеспечивающие системные вызовы, мы собрали в файле syscalls h, что позволяет нам включать его в программы этой главы. Однако имя данного файла не зафиксировано стандартом.

Параметр BUFSIZ также определен в <syscalls.h>; в каждой конкретной системе он имеет свое значение. Если размер файла не кратен BUFSIZ,

то какая-то операция чтения вернет значение меньшее, чем BUFSIZ, а следующее обращение к read даст в качестве результата нуль.

Полезно рассмотреть, как используются read и write при написании программ более высокого уровня - таких как getchar, putchar и т. д. Вот, к примеру, версия программы getchar, которая осуществляет небуферизованный ввод, читая по одному символу из стандартного входного потока.

```
#include "syscalls.h"

/* getchar: небуферизованный ввод одного символа */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

Переменная c должна быть типа char, поскольку read требует указателя на char. Приведение c к unsigned char перед тем, как вернуть ее в качестве результата, исключает какие-либо проблемы, связанные с распространением знака.

Вторая версия getchar осуществляет ввод большими кусками, но при каждом обращении выдает только один символ.

```
#include "syscalls.h"

/* getchar: простая версия с буферизацией */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* буфер пуст */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Если приведенные здесь версии функции getchar компилируются с включением заголовочного файла <stdio.h> и в этом заголовочном файле getchar определена как макрос, то нужно задать строку #undef с именем getchar.

8.3. Системные вызовы open, creat, close, unlinc

В отличие от стандартных файлов ввода, вывода и ошибок, которые открыты по умолчанию, остальные файлы нужно открывать явно. Для этого есть два системных вызова: open и creat.

Функция open почти совпадает с fopen, рассмотренной в главе 7. Разница между ними в том, что первая возвращает не файловый указатель, а дескриптор файла типа int. При любой ошибке open возвращает -1.

```
flinclude <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

Как и в fopen, аргумент name - это строка, содержащая имя файла. Второй аргумент, flags, имеет тип int и специфицирует, каким образом должен быть открыт файл. Его основными значениями являются:

```
0_RDONLY     - открыть только на чтение;
0_WRONLY     - открыть только на запись;
0_RDWR       - открыть и на чтение, и на запись.
```

В System V UNIX эти константы определены в <fcntl.h>, а в версиях Berkley (BCD) - в <sys/file.h>.

Чтобы открыть существующий файл на чтение, можно написать

```
fd = open(name, 0_RDONLY, 0);
```

Далее везде, где мы пользуемся функцией open, ее аргумент perms равен нулю.

Попытка открыть несуществующий файл является ошибкой. Создание нового файла или перезапись старого обеспечивается системным вызовом creat. Например

```
int creat(char *name, int perms);
fd = creat(name, perms);
```

Функция creat возвращает дескриптор файла, если файл создан, и -1, если по каким-либо причинам файл создать не удалось. Если файл уже существует, creat "обрежет" его до нулевой длины, что равносильно выбрасыванию предыдущего содержимого данного файла; создание уже существующего файла не является ошибкой.

Если строится действительно новый файл, то creat его создаст с правами доступа, специфицированными в аргументе perms. В системе UNIX с каждым файлом ассоциированы девять битов, содержащие информа-

цию а правах пользоваться этим файлом для чтения, записи и исполнения лицам трех категорий: собственнику файла, определенной им группе лиц и всем остальным. Таким образом, права доступа удобно специфицировать с помощью трех восьмеричных цифр. Например, 0755 специфицирует чтение, запись и право исполнения собственнику файла, а также чтение и право исполнения группе и всем остальным.

Для иллюстрации приведем упрощенную версию программы cp системы UNIX, которая копирует один файл в другой. В нашей версии копируется только один файл, не позволяется во втором аргументе указывать директорий (каталог), и права доступа не копируются, а задаются константой.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
«define PERMS 0666 /* RW для собственника, группы и остальных */

void error(char *, . . . );

/* cp: копирование f1 в f2 */
main(int argc, char *argv[])
{
  int f1, f2, n;
  char buf[BUFSIZ];

  if (argc != 3)
    error("Обращение: cp откуда куда");
  if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
    error ("cp: не могу открыть файл %s", argv[1]);
  if ((f2 = creat(argv[2], PERMS)) == -1)
    error ("cp: не могу создать файл %s, режим %03o",
      argv[2], PERMS);
  while ((n = read(f1, buf, BUFSIZ)) > 0)
    if (write(f2, buf, n) != n)
      error ("cp: ошибка при записи в файл %s", argv[2]);
  return 0;
}
```

Данная программа создает файл вывода с фиксированными правами доступа, определяемыми кодом 0666. С помощью системного вызова stat, который будет описан в параграфе 8.6, мы можем определить режим использования существующего файла и задать тот же режим для копии.

Заметим, что функция error, вызываемая с различным числом аргументов, во многом похожа на printf. Реализация error иллюстрирует, как

пользоваться другими программами семейства printf. Библиотечная функция vprintf аналогична printf, с той лишь оговоркой, что переменная часть списка аргументов заменена в ней одним аргументом, который инициализируется макросом va_start. Подобным же образом соотносятся функции vfprintf с fprintf и vsprintf с sprintf.

```
#include <stdio.h>
#include <stdarg.h>

/* error: печатает сообщение об ошибке и умирает */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "ошибка: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

На количество одновременно открытых в программе файлов имеется ограничение (обычно их число колеблется около 20). Поэтому любая программа, которая намеревается работать с большим количеством файлов, должна быть готова повторно использовать их дескрипторы. Функция close(int fd) разрывает связь между файловым дескриптором и открытым файлом и освобождает дескриптор для его применения с другим файлом. Она аналогична библиотечной функции fclose с тем лишь различием, что никакой очистки буфера не делает. Завершение программы с помощью exit или return в главной программе закрывает все открытые файлы.

Функция unlink(char *name) удаляет имя файла из файловой системы. Она соответствует функции remove стандартной библиотеки.

Упражнение 8.1. Перепишите программу cat из главы 7, используя функции read, write, open и close. Замените ими соответствующие функции стандартной библиотеки. Поэкспериментируйте, чтобы сравнить быстродействие двух версий.

8.4. Произвольный доступ (lseek)

Ввод-вывод обычно бывает последовательным, т. е. каждая новая операция чтения-записи имеет дело с позицией файла, следующей за той, что

была в предыдущей операции (чтения-записи). При желании, однако, файл можно читать или производить запись в него в произвольном порядке. Системный вызов lseek предоставляет способ передвигаться по файлу, не читая и не записывая данные. Так, функция

```
long lseek(int fd, long offset, int origin);
```

в файле с дескриптором fd устанавливает текущую позицию, смещая ее на величину offset относительно места, задаваемого значением origin. Значения параметра origin 0, 1 или 2 означают, что на величину offset отступают соответственно от начала, от текущей позиции или от конца файла. Например, если требуется добавить что-либо в файл (когда в командном интерпретаторе shell системы UNIX ввод перенаправлен оператором >> в файл или когда в fopen задан аргумент "a"), то прежде чем что-либо записывать, необходимо найти конец файла с помощью вызова функции

```
lseek(fd, 0L, 2);
```

Чтобы вернуться назад, в начало файла, надо выполнить

```
lseek(fd, 0L, ,0);
```

Следует обратить внимание на аргумент 0L: вместо 0L можно было бы написать (long) 0 или, если функция lseek должным образом объявлена, просто 0.

Благодаря lseek с файлами можно работать так, как будто это большие массивы, правда, с замедленным доступом. Например, следующая функция читает любое число байтов из любого места файла. Она возвращает число прочитанных байтов или -1 в случае ошибки.

```
#include "syscalls.h"

/* get: читает n байт из позиции pos */
int get(int fd, long pos, char *buf, int n)
{
   if (lseek(fd, pos, 0) >= 0)   /* установка позиции */
      return read(fd, buf, n);
   else
      return -1;
}
```

Возвращаемое функцией lseek значение имеет тип long и является новой позицией в файле или, в случае ошибки, равно -1. Функция fseek из стандартной библиотеки аналогична lseek; от последней она отличается тем, что в случае ошибки возвращает некоторое ненулевое значение, а ее первый аргумент имеет тип FILE*.

8.5. Пример. Реализация функций fopen и getc

Теперь на примере функций fopen и getc из стандартной библиотеки покажем, как описанные выше части согласуются друг с другом.

Напомним, что файлы в стандартной библиотеке описываются файловыми указателями, а не дескрипторами. Указатель файла - это указатель на структуру, содержащую информацию о файле: указатель на буфер, позволяющий читать файл большими кусками; число незанятых байтов буфера; указатель на следующую позицию в буфере; дескриптор файла; флажки, описывающие режим (чтение/запись), ошибочные состояния и т. д.

Структура данных, описывающая файл, содержится в <stdio.h>, который необходимо включать (с помощью #include) в любой исходный файл, если в том осуществляется стандартный ввод-вывод. Этот же заголовочный файл включен и в исходные тексты библиотеки ввода-вывода.

В следующем фрагменте, типичном для файла <stdio.h>, имена, используемые только в библиотечных функциях, начинаются с подчеркивания. Это сделано для того, чтобы они случайно не совпали с именами, фигурирующими в программе пользователя. Такое соглашение соблюдается во всех программах стандартной библиотеки.

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX 20  /* max число одновременно открытых файлов */

typedef struct _iobuf {
    int cnt;                /* количество оставшихся символов */
    char *ptr;              /* позиция следующего символа */
    char *base;             /* адрес буфера */
    int flag;               /* режим доступа */
    int fd;                 /* дескриптор файла */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin  (&iob[0])
«define stdout (&_iob[1])
#define stderr (&_iob[2])

enum _flags {
    _READ = 01,             /* файл открыт на чтение */
    _WRITE = 02,            /* файл открыт на запись */
    _UNBUF = 04,            /* файл не буферизуется */
```

```
    _EOF  = 010,         /* в данном файле встретился EOF */
    _ERR  = 020          /* в данном файле встретилась ошибка */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)     (((p)->flag & _EOF) != 0)
#define ferror(p)   (((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)   (--(p)->cnt >= 0 \
             ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
«define putc(x,p)  (--(p)->cnt >= 0 \
             ? *(p)->ptr++ = (x)  : _flushbuf((x),p))

#define getchar()  getc(stdin)
«define putchar(x) putc((x), stdout)
```

Макрос getc обычно уменьшает счетчик числа символов, находящихся в буфере, и возвращает символ, после чего приращивает указатель на единицу. (Напомним, что длинные #define с помощью обратной наклонной черты можно продолжить на следующих строках.) Когда значение счетчика становится отрицательным, getc вызывает _fillbuf, чтобы снова заполнить буфер, инициализировать содержимое структуры и выдать символ. Типы возвращаемых символов приводятся к unsigned; это гарантирует, что все они будут положительными.

Хотя в деталях ввод-вывод здесь не рассматривается, мы все же привели полное определение putc. Сделано это, чтобы показать, что она действует во многом так же, как и getc, вызывая функцию _flushbuf, когда буфер полон. В тексте имеются макросы, позволяющие получать доступ к флажкам ошибки и конца файла, а также к его дескриптору.

Теперь можно написать функцию fopen. Большая часть инструкций fopen относится к открытию файла, к соответствующему его позиционированию и к установке флажковых битов, предназначенных для индикации текущего состояния. Сама fopen не отводит места для буфера; это делает _fillbuf при первом чтении файла.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /* RW для собственника, группы и проч. */

/* fopen: открывает файл, возвращает файловый указатель */
FILE *fopen(char *name, char *mode)
```

8 Зак.1116

```
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != V && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ ! _WRITE)) == 0)
            break;              /* найдена свободная позиция*/
    if (fp >= _iob + OPEN_MAX) /* нет свободной позиции */
        return NULL;

    if (*mode== 'w')
        fd = creat(name, PERMS);
    else if (*mode== 'a') {
        if ((fd= open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, OL, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1)               /* невозможен доступ по имени name */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

Приведенная здесь версия fopen реализует не все режимы доступа, оговоренные стандартом; но, мы думаем, их реализация в полном объеме не намного увеличит длину программы. Наша fopen не распознает буквы b, сигнализирующей о бинарном вводе-выводе (поскольку в системах UNIX это не имеет смысла), и знака +, указывающего на возможность одновременно читать и писать.

Для любого файла в момент первого обращения к нему с помощью макровызова getc счетчик cnt равен нулю. Следствием этого будет вызов _fillbuf. Если выяснится, что файл на чтение не открыт, то функция _fillbuf немедленно возвратит EOF. В противном случае она попытается запросить память для буфера (если чтение должно быть с буферизацией).

После получения области памяти для буфера _fillbuf обращается к read, чтобы его наполнить, устанавливает счетчик и указатели и возвращает первый символ из буфера. В следующих обращениях _fillbuf обнаружит, что память для буфера уже выделена.

```
#include "syscalls.h"

/* _fillbuf: запрос памяти и заполнение буфера */
int _fillbuf(FILE *fp)
{
  int bufsize;

  if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
    return EOF;
  bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
  if (fp->base == NULL) /* буфера еще нет */
    if ((fp->base = (char *) malloc(bufslze)) == NULL)
      return EOF; /* нельзя получить буфер */
  fp->ptr = fp->base;
  fp->cnt = read(fp->fd, fp->ptr, bufsize);
  if (--fp->cnt < 0) {
    if (fp->cnt == -1)
      fp->flag |= _EOF;
    else
      fp->flag |= _ERR;
    fp->cnt = 0;
    return EOF;
  }
  return (unsigned char) *fp->ptr++;
}
```

Единственное, что осталось невыясненным, - это каким образом организовать начало счета. Массив _iob следует определить и инициализировать так, чтобы перед тем как программа начнет работать, в нем уже была информация о файлах stdin, stdout и stderr.

```
FILE _iob[OPEN_MAX] = { /* stdin, stdout, stderr: */
  { 0, (char *) 0, (char *) 0, _READ, 0 },
  { 0, (char *) 0, (char *) 0, _WRITE, 1 },
  { 0, (char *) 0, (char *) 0, _WRITE | _UNNBUF, 2 }
};
```

Инициализация flag как части структуры показывает, что stdin открыт на чтение, stdout - на запись, a stderr - на запись без буферизации.

Упражнение 8.2. Перепишите функции fopen и _fillbuf, работая с флажками как с полями, а не с помощью явных побитовых операций. Сравните размеры и скорости двух вариантов программ.

Упражнение 8.3. Разработайте и напишите функции _flushbuf, fflush и fclose.

Упражнение 8.4. Функция стандартной библиотеки

```
int fseek(FILE *fp, long offset, int origin)
```

идентична функции lseek с теми, однако, отличиями, что fp - это файловый указатель, а не дескриптор, и возвращает она значение int, означающее состояние файла, а не позицию в нем. Напишите свою версию fseek. Обеспечьте, чтобы работа вашей fseek по буферизации была согласована с буферизацией, используемой другими функциями библиотеки.

8.6. Пример. Печать каталогов

При разного рода взаимодействиях с файловой системой иногда требуется получить *только* информацию *о* файле, а не его содержимое. Такая потребность возникает, например, в программе печати каталога файлов, работающей аналогично команде ls системы UNIX. Она печатает имена файлов каталога и по желанию пользователя другую дополнительную информацию (размеры, права доступа и т. д.). Аналогичной командой в MS-DOS является di *г*.

Так как в системе UNIX каталог - это тоже файл, функции ls, чтобы добраться до имен файлов, нужно только его прочитать. Но чтобы получить другую информацию о файле (например узнать его размер), необходимо выполнить системный вызов. В других системах (в MS-DOS, например) системным вызовом приходится пользоваться даже для получения доступа к именам файлов. Наша цель - обеспечить доступ к информации по возможности системно-независимым способом несмотря на то, что реализация может быть существенно системно-зависима.

Проиллюстрируем сказанное написанием программы fsize. Функция fsize — частный случай программы ls; она печатает размеры всех файлов, перечисленных в командной строке. Если какой-либо из файлов сам является каталогом, то, чтобы получить информацию о нем, fsize обращается сама к себе. Если аргументов в командной строке нет, то обрабатывается текущий каталог.

Для начала вспомним структуру файловой системы в UNIXе. *Каталог* - это файл, содержащий список имен файлов и некоторую информацию о том, где они расположены. “Место расположения" - это индекс, обеспечивающий доступ в другую таблицу, называемую "списком узлов *inode*”. Для каждого файла имеется свой inode, где собрана вся информация о фай-

ле, за исключением его имени. Каждый элемент каталога состоит из двух частей: из имени файла и номера узла mode.

К сожалению, формат и точное содержимое каталога не одинаковы в разных версиях системы. Поэтому, чтобы переносимую компоненту отделить от непереносимой, разобьем нашу задачу на две. Внешний уровень определяет структуру, названную Dirent, и три подпрограммы opendir, readdir и closedir; в результате обеспечивается системно-независимый доступ к имени и номеру узла inode каждого элемента каталога. Мы будем писать программу fsize, рассчитывая на такой интерфейс, а затем покажем, как реализовать указанные функции для систем, использующих ту же структуру каталога, что и Version 7 и System V UNIX. Другие варианты оставим для упражнений.

Структура Dirent содержит номер узла inode и имя. Максимальная длина имени файла равна NAME_MAX - это значение системно-зависимо. Функция opendir возвращает указатель на структуру, названную DIR (по аналогии с FILE), которая используется функциями readdir и closedir. Эта информация сосредоточена в заголовочном файле dirent.h.

```
#define NAME_MAX 14          /* максимальная длина имени файла; */
                             /* системно-зависимая величина    */

typedef struct {             /* универс. структура элемента каталога: */
    long ino;                /* номер inode */
    char name[NAME_MAX+1];   /* имя + завершающий '\0' */
} Dirent;

typedef struct {             /* минимальный DIR: без буферизации и т.д. */
    int fd;                  /* файловый дескриптор каталога */
    Dirent d;                /* элемент каталога */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Системный вызов stat получает имя файла и возвращает полную о нем информацию, содержащуюся в узле inode, или -1 в случае ошибки. Так,

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

заполняет структуру stbuf информацией из узла inode о файле с именем

name. Структура, описывающая возвращаемое функцией stat значение, находится в <sys/stat.h> и выглядит примерно так:

```
struct stat {        /* информация из inode, возвращаемая stat */
{
   dev_t  st_dev;     /* устройство */
   ino_t  st_ino;     /* номер inode */
   short  st_mode;    /* режимные биты */
   short  st_nlink;   /* число связей с файлом */
   short  st_uid;     /* имя пользователя-собственника */
   short  st_gid;     /* имя группы собственника */
   dev_t  st_rdev;    /* для специальных файлов */
   off_t  st_size;    /* размер файла в символах */
   time_t st_atime;   /* время последнего использования */
   time_t st_mtime;   /* время последней модификации */
   time_t st_ctime;   /* время последнего изменения inode */
};
```

Большинство этих значений объясняется в комментариях. Типы, подобные dev_t и ino_t, определены в файле <sys/types.h>, который тоже нужно включить посредством #include.

Элемент st_mode содержит набор флажков, составляющих дополнительную информацию о файле. Определения флажков также содержатся в <sys/stat.h>; нам потребуется только та его часть, которая имеет дело с типом файла:

```
#define  S_IFMT   0160000   /* тип файла */
#define  S_IFDIR  0040000   /* каталог */
#define  S_IFCHR  0020000   /* символьно-ориентированный */
#define  S_IFBLK  0060000   /* блочно-ориентированный */
#define  S_IFREG  0100000   /* обычный */
```

Теперь мы готовы приступить к написанию программы fsize. Если режимные биты (st_mode), полученные от stat, указывают, что файл не является каталогом, то можно взять его размер (st_size) и напечатать. Однако если файл - каталог, то мы должны обработать все его файлы, каждый из которых в свою очередь может быть каталогом. Обработка каталога - процесс рекурсивный.

Программа main просматривает параметры командной строки, передавая каждый аргумент функции fsize.

```
«include <stdio.h>,
#include<string.h>
#include "syscalls. h"
«include <fcntl.h>        /* флажки чтения и записи */
#include <sys/types.h>    /* определения типов */
```

```
#include <sys/stat.h>    /* структура, возвращаемая stat */
#include "dirent.h"

void fsize(char *);

/* печатает размеры файлов */
main(int argc, char **argv)
{
  if (argc == 1) /* по умолчанию берется текущий каталог */
    fsize(".");
  else
    while (--argc > 0)
      fsize(*++argv);
  return 0;
}
```

Функция fsize печатает размер файла. Однако, если файл - каталог, она сначала вызывает dirwalk, чтобы обработать все его файлы. Обратите внимание на то, как используются имена флажков S_IFMT и S_IFDIR из <sys/stat.h> при проверке, является ли файл каталогом. Здесь нужны скобки, поскольку приоритет оператора & ниже приоритета оператора ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: печатает размер файла "name" */
void fsize(char *name)
{
  struct stat stbuf;

  if (stat(name, &stbuf) == -1) {
    fprintf(stderr, "fsize: нет доступа к %s\n", name);
    return;
  }
  if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
    dirwalk(name, fsize);
  printf("%8ld %s\n", stbuf.st_size, name);
}
```

Функция dirwalk - это универсальная программа, применяющая некоторую функцию к каждому файлу каталога. Она открывает каталог, с помощью цикла перебирает содержащиеся в нем файлы, применяя к каждому из них указанную функцию, затем закрывает каталог и осуществляет возврат. Так как fsize вызывает dirwalk на каждом каталоге, в этих двух функциях заложена косвенная рекурсия.

```
#define MAX_PATH 1024

/* dirwalk: применяет fcn ко всем файлам из dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: не могу открыть %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
            || strcmp(dp->name, "..") == 0)
                continue;        /* пропустить себя и родителя */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
                fprintf(stderr, "dirwalk: слишком длинное имя %s/%s\n",
                    dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

Каждый вызов readdir возвращает указатель на информацию о следующем файле или NULL, если все файлы обработаны. Любой каталог всегда хранит в себе информацию о себе самом в файле под именем "." и о своем родителе в файле под именем ".."; их нужно пропустить, иначе программа зациклится. Обратите внимание: код программы этого уровня не зависит от того, как форматированы каталоги. Следующий шаг — представить минимальные версии opendir, readdir и closedir для некоторой конкретной системы. Здесь приведены программы для систем Version 7 и System V UNIX. Они используют информацию о каталоге, хранящуюся в заголовочном файле <sys/dir.h>, который выглядит следующим образом:

```
#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct direct          /* элемент каталога */
{
    ino_t d_ino;       /* номер inode */
```

```
    char d_name[DIRSIZ]; /* длинное имя не имеет '\0' */
};
```

Некоторые версии системы допускают более длинные имена и имеют более сложную структуру каталога.

Тип ino_t задан с помощью typedef и описывает индекс списка узлов mode. В системе, которой пользуемся мы, этот тип есть unsigned short, но в других системах он может быть иным, поэтому его лучше определять через typedef. Полный набор "системных" типов находится в <sys/types.h>.

Функция opendir открывает каталог, проверяет, является ли он действительно каталогом (в данном случае это делается с помощью системного вызова fstat, который аналогичен stat, но применяется к дескриптору файла), запрашивает пространство для структуры каталога и записывает информацию.

```
int fstat(int fd, struct stat *);

/* opendir: открывает каталог для вызовов readdir */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
        || fstat(fd, &stbuf) == -1
        || (stbuf.st_mode & S_IFMT) != S_IFDIR
        || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
          return NULL;
    dp->fd = fd;
    return dp;
}
```

Функция closedir закрывает каталог и освобождает пространство.

```
/* closedir: закрывает каталог, открытый opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Наконец, readdir с помощью read читает каждый элемент каталога. Если некий элемент каталога в данный момент не используется (соответствующий ему файл был удален), то номер узла mode у него равен нулю, и данная позиция пропускается. В противном случае номер mode и имя размещаются в статической (static) структуре, и указатель на нее выдается в качестве результата. При каждом следующем обращении новая информация занимает место предыдущей.

```
#include <sys/dir.h>     /* место расположения структуры каталога */

/* readdir: последовательно читает элементы каталога */
Dirent *readdir(DIR *dp)
{
  struct direct dirbuf;   /* структура каталога на данной системе */
  static Dirent d;       /* возвращает унифицированную структуру */

  while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
          == sizeof(dirbuf)) {
    if (dirbuf.d_ino == 0)  /* пустой элемент, не используется */
      continue;
    d.ino = dirbuf.d_ino;
    strncpy(d.name, dirbuf.d_name, DIRSIZ);
    d.name[DIRSIZ] = '\0';              /* завершающий символ '\0' */
    return &d;
  }
  return NULL;
}
```

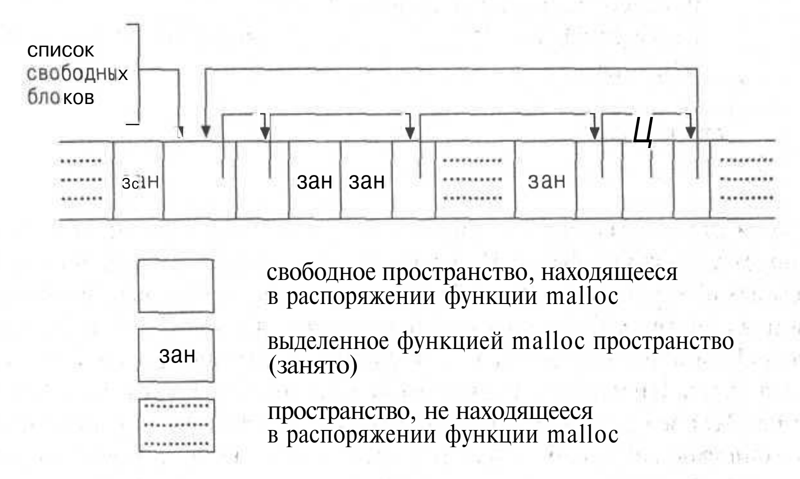
Хотя программа fsize - довольно специализированная, она иллюстрирует два важных факта. Первый: многие программы не являются "системными"; они просто используют информацию, которую хранит операционная система. Для таких программ существенно то, что представление информации сосредоточено исключительно в стандартных заголовочных файлах. Программы включают эти файлы, а не держат объявления в себе. Второе наблюдение заключается в том, что при старании системно-зависимым объектам можно создать интерфейсы, которые сами не будут системно-зависимыми. Хорошие тому примеры - функции стандартной библиотеки.

Упражнение 8.5. Модифицируйте fsize таким образом, чтобы можно было печатать остальную информацию, содержащуюся в узле mode,

8.7. Пример. Распределитель памяти

В главе 5 был описан простой распределитель памяти, основанный на принципе стека. Версия, которую мы напишем здесь, не имеет ограничений: вызовы `malloc` и `free` могут выполняться в любом порядке; malloc делает запрос в операционную систему на выделение памяти тогда, когда она требуется. Эти программы иллюстрируют приемы, позволяющие получать машинно-зависимый код сравнительно машинно-независимым способом, и, кроме того, они могут служить примером применения таких средств языка, как структуры, объединения И `typedef`.

Никакого ранее скомпилированного массива фиксированного размера, из которого выделяются куски памяти, не будет. Функция malloc запрашивает память у операционной системы по мере надобности. Поскольку и другие действия программы могут вызывать запросы памяти, которые удовлетворяются независимо от этого распределителя памяти, пространство, которым заведует malloc, не обязательно представляет собой связный кусок памяти. Поэтому свободная память хранится в виде списка блоков. Каждый блок содержит размер, указатель на следующий блок и само пространство. Блоки в списке хранятся в порядке возрастания адресов памяти, при этом последний блок (с самым большим адресом) указывает на первый.



При возникновении запроса на память просматривается список свободных блоков, пока не обнаружится достаточно большой блок. Такой алгоритм называется "поиском первого подходящего" в отличие от алгоритма "поиска наилучшего подходящего", который ищет наименьший блок из числа удовлетворяющих запросу. Если размер блока в точности

соответствует требованиям, то такой блок исключается из списка и отдается в пользование. Если размер блока больше, чем требуется, от него отрезается нужная часть - она отдается пользователю, а ненужная оставляется в списке свободных блоков. Если блока достаточного размера не оказалось, то у операционной системы запрашивается еще один большой кусок памяти, который присоединяется к списку свободных блоков.

Процедура освобождения сопряжена с прохождением по списку свободных блоков, поскольку нужно найти подходящее место для освобождаемого блока. Если подлежащий освобождению блок примыкает с какой-то стороны к одному из свободных блоков, то он объединяется с ним в один блок большего размера, чтобы по возможности уменьшить раздробленность (фрагментацию) памяти. Выполнение проверки, примыкают ли блоки друг к другу, не составляет труда, поскольку список свободных блоков всегда упорядочен по возрастанию адресов.

Существует проблема, о которой мы уже упоминали в главе 5, состоящая в том, что память, выдаваемая функцией malloc, должна быть соответствующим образом выровнена с учетом объектов, которые будут в ней храниться. Хотя машины и отличаются друг от друга, но для каждой из них существует тип, предъявляющий самые большие требования на выравнивание, и, если по некоему адресу допускается размещение объекта этого типа, то по нему можно разместить и объекты всех других типов. На некоторых машинах таким самым "требовательным" типом является double, на других это может быть int или long.

Свободный блок содержит указатель на следующий блок в списке, свой размер и собственно свободное пространство. Указатель и размер представляют собой управляющую информацию и образуют так называемый "заголовок". Чтобы упростить выравнивание, все блоки создаются кратными размеру заголовка, а заголовок соответствующим образом выравнивается. Этого можно достичь, сконструировав объединение, которое будет содержать соответствующую заголовку структуру и самый требовательный в отношении выравнивания тип. Для конкретности мы выбрали тип long.

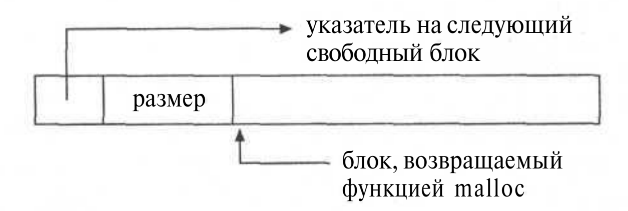
```
typedef long Align;      /* для выравнивания по границе long */

union header {           /* заголовок блока: */
    struct {
        union header *ptr; /* след. блок в списке свободных */
        unsigned size;     /* размер этого блока */
    } s;
    Align x;             /* принудительное выравнивание блока */
};

typedef union header Header;
```

Поле Align нигде не используется; оно необходимо только для того, чтобы каждый заголовок был выровнен по самому "худшему" варианту границы.

Затребованное число символов округляется в malloc до целого числа единиц памяти размером в заголовок (именно это число и записывается в поле size (размер) в заголовке); кроме того, в блок входит еще одна единица памяти - сам заголовок. Указатель, возвращаемый функцией malloc, указывает на свободное пространство, а не на заголовок. Со свободным пространством пользователь может делать что угодно, но, если он будет писать что-либо за его пределами, то, вероятно, список разрушится.



Поскольку память, управляемая функцией malloc, не обладает связностью, размеры блоков нельзя вычислить по указателям, и поэтому без поля, хранящего размер, нам не обойтись.

Для организации начала работы используется переменная base. Если freep есть NULL (как это бывает при первом обращении к malloc), создается "вырожденный" список свободного пространства; он содержит один блок нулевого размера с указателем на самого себя. Поиск свободного блока подходящего размера начинается с этого указателя (freep), т. е. с последнего найденного блока; такая стратегия помогает поддерживать список однородным. Если найденный блок окажется слишком большим, пользователю будет отдана его хвостовая часть; при этом потребуется только уточнить его размер в заголовке найденного свободного блока. В любом случае возвращаемый пользователю указатель является адресом свободного пространства, размещающегося в блоке непосредственно за заголовком.

```
static Header base;            /* пустой список для нач. запуска */
static Header *freep = NULL;   /* начало в списке своб. блоков */

/* malloc: универсальный распределитель памяти */
void *malloc(unsigned nbytes)
{
   Header *p, *prevp;
   Header *morecore(unsigned);
   unsigned nunits;
```

```
    nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* списка своб. памяти еще нет */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {       /* достаточно большой */
            if (p->s.size == nunits)     /* точно нужного размера */
                prevp->s.ptr = p->s.ptr;
            else {    /* отрезаем хвостовую часть */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)                 /* прошли полный цикл по списку */
            if ((p = morecore(nunits)) == NULL)
                return NULL;             /* больше памяти нет */
    }
}
```

Функция morecore получает память от операционной системы. Детали того, как это делается, могут не совпадать в различных системах. Так как запрос памяти у системы - сравнительно дорогая операция, мы бы не хотели для этого каждый раз обращаться к malloc. Поэтому используется функция morecore, которая запрашивает не менее NALLOC единиц памяти; этот больший кусок памяти будет "нарезаться" потом по мере надобности. После установки в поле размера соответствующего значения функция morecore вызывает функцию free и тем самым включает полученный кусок в список свободных областей памяти.

```
#define NALLOC 1024 /* миним. число единиц памяти для запроса */

/* morecore: запрашивает у системы дополнительную память */
static Header * morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
```

```
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)          /* больше памяти нет */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

Системный вызов sbrk(n) в UNIXe возвращает указатель на n байт памяти или -1, если требуемого пространства не оказалось, хотя было бы лучше, если бы в последнем случае он возвращал NULL. Константу -1 необходимо привести ктипу char *, чтобы ее можно было сравнить с возвращаемым значением. Это еще один пример того, как операция приведения типа делает функцию относительно независимой от конкретного представления указателей на различных машинах. Есть, однако, одна "некорректность", состоящая в том, что сравниваются указатели на различные блоки, выдаваемые функцией sbrk. Такое сравнение не гарантировано стандартом, который позволяет сравнивать указатели лишь в пределах одного и того же массива. Таким образом, эта версия malloc верна только на тех машинах, в которых допускается сравнение любых указателей.

В заключение рассмотрим функцию free. Она просматривает список свободной памяти, начиная с freep, чтобы подыскать место для вставляемого блока. Искомое место может оказаться или между блоками, или в начале списка, или в его конце. В любом случае, если подлежащий освобождению блок примыкает к соседнему блоку, он объединяется с ним в один блок. О чем еще осталось позаботиться, - так это о том, чтобы указатели указывали в нужные места и размеры блоков были правильными.

```
/* free: включает блок в список свободной памяти */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap -1;          /* указатель на заголовок блока */
    for (p=freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;   /* освобождаем блок в начале или в конце */

    if (bp + bp->s.size == p->s.ptr) {   /* слить с верхним */
        bp->s.size += p->s.ptr->s.size;    /* соседом */
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
```

```
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {      /* слить с нижним соседом */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Хотя выделение памяти по своей сути - машинно-зависимая проблема, с ней можно справиться, что и иллюстрирует приведенная программа, в которой машинная зависимость упрятана в очень маленькой ее части. Что касается проблемы выравнивания, то мы разрешили ее с помощью typedef и union (предполагается, что sbrk дает подходящий в смысле выравнивания указатель). Операции приведения типов позволяют нам сделать явными преобразования типов и даже справиться с плохо спроектированным интерфейсом системы. Несмотря на то, что наши рассуждения касались распределения памяти, этот общий подход применим и в других ситуациях.

Упражнение 8.6. Стандартная функция calloc(n, size) возвращает указатель на n элементов памяти размера size, заполненных нулями. Напишите свой вариант calloc, пользуясь функцией malloc или модифицируя последнюю.

Упражнение 8.7. Функция malloc допускает любой размер, никак не проверяя его на правдоподобие; free предполагает, что размер освобождаемого блока — правильный. Усовершенствуйте эти программы таким образом, чтобы они более тщательно контролировали ошибки.

Упражнение 8.8. Напишите программу bfree(p,n), освобождающую произвольный блок p, состоящий из n символов, путем включения его в список свободной памяти, поддерживаемый функциями malloc и free. С помощью bfree пользователь должен иметь возможность в любое время добавить в список свободной памяти статический или внешний массив.

Справочное руководство

А 1. Введение

Данное руководство описывает язык программирования Си, определенный 31 октября 1989 г. в соответствии с проектом, утвержденным в ANSI в качестве Американского национального стандарта для информационных систем: Язык программирования Си, X3.159–1989 ("American National Standard for Information Systems - Programming Language C, X3.159–1989"). Это описание - лишь один из вариантов предлагаемого стандарта, а не сам стандарт, однако мы специально заботились о том, чтобы сделать его надежным руководством по языку.

Настоящий документ в основном следует общей схеме описания, принятой в стандарте (публикация которого в свою очередь основывалась на первом издании этой книги), однако в организационном плане есть различия. Если не считать отклонений в названиях нескольких продуктов и отсутствия формальных определений лексем и препроцессора, грамматика языка здесь и грамматика в стандарте эквивалентны.

Далее примечания (как и это) набираются с отступом от левого края страницы. В основном эти примечания касаются отличий стандарта от версии языка, описанной в первом издании этой книги, и от последующих нововведений в различных компиляторах.

А2. Соглашения о лексике

Программа состоит из одной или нескольких *единиц трансляции,* хранящихся в виде файлов. Каждая такая единица проходит несколько фаз трансляции, описанных в А12. Начальные фазы осуществляют лексические преобразования нижнего уровня, выполняют директивы, заданные

в программе строками, начинающимися со знака #, обрабатывают макроопределения и производят макрорасширения. По завершении работы препроцессора (А12) программа представляется в виде последовательности лексем.

А 2.1. Лексемы (tokens)

Существуют шесть классов лексем (или токенов): идентификаторы, ключевые слова, константы, строковые литералы, операторы и прочие разделители. Пробелы, горизонтальные и вертикальные табуляции, новые строки, переводы страницы и комментарии (имеющие общее название символы-разделители) рассматриваются компилятором только как разделители лексем и в остальном на результат трансляции влияния не оказывают. Любой из символов-разделителей годится, чтобы отделить друг от друга соседние идентификаторы, ключевые слова и константы.

Если входной поток уже до некоторого символа разбит на лексемы, то следующей лексемой будет самая длинная строка, которая может быть лексемой.

А 2.2. Комментарий

Символы /* открывают комментарий, а символы */ закрывают его. Комментарии нельзя вкладывать друг в друга, их нельзя помещать внутрь строк или текстовых литералов.

А 2.3. Идентификаторы

Идентификатор — это последовательность букв и цифр. Первым символом должна быть буква; знак подчеркивания _ считается буквой. Буквы нижнего и верхнего регистров различаются. Идентификаторы могут иметь любую длину; для внутренних идентификаторов значимыми являются первые 31 символ; в некоторых реализациях принято большее число значимых символов. К внутренним идентификаторам относятся имена макросов и все другие имена, не имеющие внешних связей (А11.2). На идентификаторы с внешними связями могут накладываться большие ограничения: иногда воспринимаются не более шести первых символов и могут не различаться буквы верхнего и нижнего регистров.

А 2.4. Ключевые слова

Следующие идентификаторы зарезервированы в качестве ключевых слов и в другом смысле использоваться не могут:

```
auto      char      default   else
break     const     do        enum
case      continue  double    extern
```

```
float     long       sizeof     union
for       register   static     unsigned
goto      return     struct     void
if        short      switch     volatile
int       signed     typedef    while
```

В некоторых реализациях резервируются также слова fortran и asm.

Ключевые слова const, signed и volatile впервые появились в стандарте ANSI; enum и void - новые по отношению к первому изданию книги, но уже использовались; ранее зарезервированное entry нигде не использовалось и поэтому более не резервируется.

А 2.5. Константы

Существует несколько видов констант. Каждая имеет свой тип данных; базовые типы рассматриваются в А4.2.

константа:

целая-константа
символьная-константа
константа-с-плавающей-точкой
константа-перечисление

А 2.5.1. Целые константы

Целая константа, состоящая из последовательности цифр, воспринимается как восьмеричная, если она начинается с 0 (цифры нуль), и как десятичная в противном случае. Восьмеричная константа не содержит цифр 8 и 9. Последовательность цифр, перед которой стоят Ox или OX, рассматривается как шестнадцатеричное целое. В шестнадцатеричные цифры включены буквы от a (или A) до f (или F) со значениями от 10 до 15.

Целая константа может быть записана с буквой-суффиксом u (или U) для спецификации ее как беззнаковой константы. Она также может быть с буквой-суффиксом 1 (или L) для указания, что она имеет тип long.

Тип целой константы зависит от ее вида, значения и суффикса (о типах см. А4). Если константа - десятичная и не имеет суффикса, то она принимает первый из следующих типов, который годится для представления ее значения: int, long int, unsigned long int. Восьмеричная или шестнадцатеричная константа без суффикса принимает первый возможный из типов: int, unsigned int, long int, unsigned long int. Если константа имеет суффикс u или U, то она принимает первый возможный из типов: unsigned int, unsigned long int. Если константа имеет суффикс 1 или L, то она принимает первый возможный из типов: long int, unsigned long int. Если константа имеет суффикс ul или UL, то она принимает тип unsigned long int.

Типы целых констант получили существенное развитие в сравнении с первой редакцией языка, в которой большие целые имели просто тип `long`. Суффиксы U и и введены впервые.

А 2.5.2. Символьные константы

Символьная константа - это последовательность из одной или нескольких символов, заключенная в одиночные кавычки (например 'х'). Если внутри одиночных кавычек расположен один символ, значением константы является числовое значение этого символа в кодировке, принятой на данной машине. Значение константы с несколькими символами зависит от реализации.

Символьная константа не может содержать в себе одиночную кавычку ' или символ новой строки; чтобы изобразить их и некоторые другие символы, могут быть использованы эскейп-последовательности:

новая строка (newline, linefeed)	NL (LF)	\n
горизонтальная табуляция (horisontal tab)	HT	\t
вертикальная табуляция (vertical tab)	VT	\v
возврат на шаг (backspace)	BS	\b
возврат каретки (carriage return)	CR	\r
перевод страницы (formfeed)	FF	\f
сигнал звонок (audible alert, bell)	BEL	\a
обратная наклонная черта (backslash)	\	\\
знак вопроса (question mark)	?	\?
одиночная кавычка (single quote)		\'
двойная кавычка (double quote)		\"
восьмеричный код (octal number)	*ooo*	*ooo*
шестнадцатеричный код (hex number)	*hh*	\x*hh*

Эскейп-последовательность *ooo* состоит из обратной наклонной черты, за которой следуют одна, две или три восьмеричные цифры, специфицирующие значение желаемого символа. Наиболее частым примером такой конструкции является `\0` (за которой не следует цифра); она специфицирует `NULL`-символ. Эскейп-последовательность \x*hh* состоит из обратной наклонной черты с буквой x, за которыми следуют шестнадцатеричные цифры, специфицирующие значение желаемого символа. На количество цифр нет ограничений, но результат будет не определен, если значение полученного символа превысит значение самого "большого" из допустимых символов. Если в данной реализации тип `char` трактуется как число

со знаком, то значение и в восьмеричной, и в шестнадцатеричной эскейп-последовательности получается с помощью "распространения знака", как если бы выполнялась операция приведения к типу char. Если за \ не следует ни один из перечисленных выше символов, результат не определен.

В некоторых реализациях имеется расширенный набор символов, который не может быть охвачен типом char. Константа для такого набора пишется с буквой L впереди (например L'x') и называется расширенной символьной константой. Такая константа имеет тип wchar_t (целочисленный тип, определенный в стандартном заголовочном файле <stddef.h>). Как и в случае обычных символьных констант, здесь также возможны восьмеричные и шестнадцатеричные эскейп-последовательности; если специфицированное значение превысит тип wchar_t, результат будет не определен.

Некоторые из приведенных эскейп-последовательностей новые (шестнадцатеричные в частности). Новым является и расширенный тип для символов. Наборам символов, обычно используемым в Америке и Западной Европе, подходит тип char, а тип wchar_t был добавлен главным образом для азиатских языков.

А 2.5.3. Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, дробной части, e или E и целого (возможно, со знаком), представляющего порядок, и, возможно, суффикса типа, задаваемого одной из букв: f, F, l или L. И целая, и дробная часть представляют собой последовательность цифр. Либо целая часть, либо дробная часть (но не обе вместе) могут отсутствовать; также могут отсутствовать десятичная точка или E с порядком (но не обе одновременно). Тип определяется суффиксом; F или f определяют тип float, L или l – тип long double; при отсутствии суффикса подразумевается тип double.

Суффиксы для констант с плавающей точкой являются нововведением.

А 2.5.4. Константы-перечисления

Идентификаторы, объявленные как элементы перечисления (А8.4), являются константами типа int.

А 2.6. Строковые литералы

Строковый литерал, который также называют строковой константой, - это последовательность символов, заключенная в двойные кавычки (Например, "..."). Строка имеет тип "массив символов" и память класса static (А4), которая инициализируется заданными символами. Представляются ли одинаковые строковые литералы одной копией или несколькими,

зависит от реализации. Поведение программы, пытающейся изменить строковый литерал, не определено.

Написанные рядом строковые литералы объединяются (конкатенируются) в одну строку. После любой конкатенации к строке добавляется NULL-байт (\0), что позволяет программе, просматривающей строку, найти ее конец. Строковые литералы не могут содержать в себе символ новой строки или двойную кавычку; в них нужно использовать те же эскейп-последовательности, что и в символьных константах.

Как и в случае с символьными константами, строковый литерал с символами из расширенного набора должен начинаться с буквы L (например L"..."). Строковый литерал из расширенного набора имеет тип "массив из wchar_t". Конкатенация друг с другом обычных и "расширенных" строковых литералов не определена.

То, что строковые литералы не обязательно представляются разными копиями, запрет на их модификацию, а также конкатенация соседних строковых литералов - нововведения ANSI-стандарта. "Расширенные" строковые литералы также объявлены впервые.

А 3. Нотация синтаксиса

В нотации синтаксиса, используемой в этом руководстве, синтаксические понятия набираются курсивом, а слова и символы, воспринимаемые буквально, обычным шрифтом. Альтернативные конструкции обычно перечисляются в столбик (каждая альтернатива на отдельной строке); в редких случаях длинные списки небольших по размеру альтернатив располагаются в одной строке, помеченной словами "один из". Необязательное слово-термин или не термин снабжается индексом *"необ."*.Так, запись

{ *выражение*$_{необ}$ }

обозначает выражение, заключенное в фигурные скобки, которое в общем случае может отсутствовать. Полный перечень синтаксических конструкций приведен в А13.

В отличие от грамматики, данной в первом издании этой книги, приведенная здесь грамматика старшинство и порядок выполнения операций в выражениях описывает явно.

А4. Что обозначают идентификаторы

Идентификаторы, или имена, ссылаются на разные объекты (в оригинале - *things.* - *Примеч. ред.):* функции; теги структур, объединений и пере-

числений; элементы структур или объединений; typedef-имена; метки и объекты. Объектом (называемым иногда переменной) является часть памяти, интерпретация которой зависит от двух главных характеристик: *класса памяти* и ее *типа*. Класс памяти сообщает о времени жизни памяти, связанной с идентифицируемым объектом; тип определяет, какого рода значения находятся в объекте. С любым именем ассоциируются своя область видимости (т. е. тот участок программы, где это имя известно) и атрибут связи, определяющий, обозначает ли это имя в другом файле тот же самый объект или функцию. Область видимости и атрибут связи обсуждаются в А11.

А 4.1. Класс памяти

Существуют два класса памяти: автоматический и статический. Несколько ключевых слов в совокупности с контекстом объявлений объектов специфицируют класс памяти для этих объектов.

Автоматические объекты локальны в блоке (А9.3), при выходе из него они "исчезают". Объявление, заданное внутри блока, если в нем отсутствует спецификация класса памяти или указан спецификатор auto, создает автоматический объект. Объект, помеченный в объявлении словом register, является автоматическим и размещается по возможности в регистре машины.

Статические объекты могут быть локальными в блоке или располагаться вне блоков, но в обоих случаях их значения сохраняются после выхода из блока (или функции) до повторного *в* него входа. Внутри блока (в том числе и в блоке, образующем тело функции) статические объекты в объявлениях помечаются словом static. Объекты, объявляемые вне всех блоков на одном уровне с определениями функций, всегда статические. С помощью ключевого слова static их можно сделать локальными в пределах транслируемой единицы (в этом случае они получают атрибут *внутренней связи), и* они становятся глобальными для всей программы, если опустить явное указание класса памяти или использовать ключевое слово extern (в этом случае они получают атрибут *внешней связи).*

А 4.2. Базовые типы

Существует несколько базовых типов. Стандартный заголовочный файл <limits.h>, описанный в приложении В, определяет самое большое и самое малое значения для каждого типа в данной конкретной реализации. В приложении В приведены минимально возможные величины.

Размер объектов, объявляемых как символы, позволяет хранить любой символ из набора символов, принятого в машине. Если объект типа char действительно хранит символ из данного набора, то его значением является код этого символа, т. е. некоторое неотрицательное целое. Переменные типа char могут хранить и другие значения, но тогда диапазон их

значений и особенно вопрос о том, знаковые эти значения или беззнаковые, зависит от реализации.

Беззнаковые символы, объявленные с помощью слов unsigned char, имеют ту же разрядность, что и обычные символы, но представляют неотрицательные значения; с помощью слов signed char можно явно объявить символы со знаком, которые занимают столько же места, как и обычные символы.

> Тип unsigned char не упоминался в первой редакции языка, но всеми использовался. Тип signed char - новый.

Помимо char среди целочисленных типов могут быть целые трех размеров: short int, int и long int. Обычные объекты типа int имеют естественный размер, принятый в архитектуре данной машины, другие размеры предназначены для специальных нужд. Более длинные целые по крайней мере покрывают все значения более коротких целых, однако в некоторых реализациях обычные целые могут быть эквивалентны коротким (short) или длинным (long) целым. Все типы int представляют значения со знаком, если не оговорено противное.

Для беззнаковых целых в объявлениях используется ключевое слово unsigned. Такие целые подчиняются арифметике по модулю 2^n, где *n* - число битов в представлении числа, и, следовательно, в арифметике с беззнаковыми целыми никогда не бывает переполнения. Множество неотрицательных значений, которые могут храниться в объектах со знаком, является подмножеством значений, которые могут храниться в соответствующих объектах без знака; знаковое и беззнаковое представления каждого такого значения совпадают.

Любые два из типов с плавающей точкой: с одинарной точностью (float), с двойной точностью (double) и с повышенной точностью (long double) могут быть синонимами, но каждый следующий тип этого списка должен по крайней мере обеспечивать точность предыдущего.

> long double - новый тип. В первой редакции языка синонимом для double был long float, теперь последний изъят из обращения.

Перечисления - единственные в своем роде типы, которым дается полный перечень значений; с каждым перечислением связывается множество именованных констант (А8.4). Перечисления ведут себя наподобие целых, но компилятор обычно выдает предупреждающее сообщение, если объекту некоторого перечислимого типа присваивается нечто, отличное от его константы, или выражение не из этого перечисления.

Поскольку объекты перечислений можно рассматривать как числа, перечисление относят к *арифметическому* типу. Типы char и int всех размеров, каждый из которых может быть со знаком или без знака, а также

перечисления называют *целочисленными (integral)* типами. Типы float, double и long double называются типами с *плавающей точкой.*

Тип void специфицирует пустое множество значений. Он используется как "тип возвращаемого функцией значения" в том случае, когда она не генерирует никакого результирующего значения.

А 4.3. Производные типы

Помимо базовых типов существует практически бесконечный класс производных типов, которые формируются из уже существующих и описывают следующие конструкции:

- *массивы* объектов заданного типа;
- *функции,* возвращающие объекты заданного типа;
- *указатели* на объекты заданного типа;
- *структуры,* содержащие последовательность объектов, возможно, различных заданных типов;
- *объединения,* каждое из которых может содержать любой из нескольких объектов различных заданных типов.

В общем случае приведенные методы конструирования объектов могут применяться рекурсивно.

А 4.4. Квалификаторы типов

Тип объекта может снабжаться квалификатором. Объявление объекта с квалификатором const указывает на то, что его значение далее не будет изменяться; объявляя объект как volatile (изменчивый, непостоянный (*англ.*)), мы указываем на его особые свойства для выполняемой компилятором оптимизации. Ни один из квалификаторов на диапазоны значений и арифметические свойства объектов не влияет. Квалификаторы обсуждаются в А8.2.

А 5. Объекты и Lvalues

Объект - это некоторая именованная область памяти; *lvalue* - это выражение, обозначающее объект. Очевидным примером lvalue является идентификатор с соответствующим типом и классом памяти. Существуют операции, порождающие lvalue. Например, если Е - выражение типа указатель, то * Е есть выражение для lvalue, обозначающего объект, на который указывает Е. Термин "lvalue" произошел от записи присваивания E1 = E2, в которой левый *(left* - левый *(англ.),* отсюда буква l, *value* - значение) операнд E1 должен быть выражением lvalue. Описывая каждый оператор, мы сообщаем, ожидает ли он lvalue в качестве операндов и выдает ли lvalue в качестве результата.

А6. Преобразования

Некоторые операторы в зависимости от своих операндов могут вызывать преобразование их значений из одного типа в другой. В этом параграфе объясняется, что следует ожидать от таких преобразований. В А6.5 формулируются правила, по которым выполняются преобразования для большинства обычных операторов. При рассмотрении каждого отдельного оператора эти правила могут уточняться.

А 6.1. Целочисленное повышение

Объект типа перечисление, символ, короткое целое, целое в битовом поле - все они со знаком или без могут использоваться в выражении там, где возможно применение целого. Если тип int позволяет "охватить" все значения исходного типа операнда, то операнд приводится к int, в противном случае он приводится к unsigned int. Эта процедура называется *целочисленным повышением*[1].

А 6.2. Целочисленные преобразования

Любое целое приводится к некоторому заданному беззнаковому типу путем поиска конгруэнтного (т. е. имеющего то же двоичное представление) наименьшего неотрицательного значения и получения остатка от деления его на *nmax + 1,* где *nmax* - наибольшее число в этом беззнаковом типе. Для двоичного представления в дополнительном коде это означает либо выбрасывание лишних старших разрядов, если беззнаковый тип "уже" исходного типа, либо заполнение недостающих старших разрядов нулями (для значения без знака) или значением знака (для значения со знаком), если беззнаковый тип "шире" исходного.

В результате приведения любого целого к знаковому типу преобразуемое значение не меняется, если оно представимо в этом новом типе, в противном случае результат зависит от реализации.

А 6.3. Целые и числа с плавающей точкой

При преобразовании из типа с плавающей точкой в целочисленный дробная часть значения отбрасывается; если полученное при этом значение нельзя представить в заданном целочисленном типе, то результат не определен. В частности, не определен результат преобразования отрицательных значений с плавающей точкой в беззнаковые целые.

Если значение преобразуется из целого в величину с плавающей точкой и она находится в допустимом диапазоне, но представляется в новом

---

[1] *Integral promotion* - целочисленное повышение — иногда также переводят как "интегральное продвижение". - *Примеч. ред.*

типе неточно, то результатом будет одно из двух значений нового типа, ближайших к исходному. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

А 6.4. Типы с плавающей точкой

При преобразовании из типа с плавающей точкой меньшей точности в тип с плавающей точкой большей точности значение не изменяется. Если, наоборот, переход осуществляется от большей точности к меньшей и значение остается в допустимых пределах нового типа, то результатом будет одно из двух ближайших значений нового типа. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

А 6.5. Арифметические преобразования

Во многих операциях преобразование типов операндов и определение типа результата осуществляются по одним и тем же правилам. Они состоят в том, что операнды приводятся к некоторому общему типу, который также является и типом результата. Эти правила называются *обычными арифметическими преобразованиями.*

- Если какой-либо из операндов имеет тип long double, то другой приводится к long double.
- В противном случае, если какой-либо из операндов имеет тип double, то другой приводится к double.
- В противном случае, если какой-либо из операндов имеет тип float, то другой приводится к float.
- В противном случае для обоих операндов осуществляется целочисленное повышение; затем, если один из операндов имеет тип unsigned long int, то и другой преобразуется в unsigned long int.
- В противном случае, если один из операндов принадлежит типу long int, а другой - unsigned int, то результат зависит от того, покрывает ли long int все значения unsigned int, и если это так, то unsigned int приводится к long int; если нет, то оба операнда преобразуются в unsigned long int.
- В противном случае, если один из операндов имеет тип long int, то другой приводится к long int.
- В противном случае, если один из операндов - unsigned int, то другой приводится к unsigned int.
- В противном случае оба операнда имеют тип int.

Здесь есть два изменения. Во-первых, арифметика с операндами с плавающей точкой теперь может производиться с одинарной точностью, а не только с двойной; в первой редакции языка вся арифметика с плавающей точкой производилась с двойной точностью. Во-вторых, более короткий беззнаковый

тип в комбинации с более длинным знаковым типом не распространяет свойство беззнаковости на тип результата; в первой редакции беззнаковый тип всегда доминировал. Новые правила немного сложнее, но до некоторой степени уменьшают вероятность появления неожиданных эффектов в комбинациях знаковых и беззнаковых величин. При сравнении беззнакового выражения со знаковым того же размера все же может возникнуть неожиданный результат.

А 6.6. Указатели и целые

К указателю можно прибавлять (и вычитать из него) выражение целочисленного типа; последнее в этом случае подвергается преобразованию, описанному в А7.7 при рассмотрении оператора сложения.

К двум указателям на объекты одного типа, принадлежащие одному массиву, может применяться операция вычитания; результат приводится к целому посредством преобразования, описанного в А7.7 при рассмотрении оператора вычитания.

Целочисленное константное выражение со значением 0 или оно же, но приведенное к типу void *, может быть преобразовано в указатель любого типа операторами приведения, присваивания и сравнения. Результатом будет NULL-указатель, который равен любому другому NULL-указателю того же типа, но не равен никакому указателю на реальный объект или функцию.

Для указателей допускаются и другие преобразования, но в связи с ними возникает проблема зависимости результата от реализации. Эти преобразования должны быть специфицированы явным оператором преобразования типа или оператором приведения (А7.5 и А8.8).

Указатель можно привести к целочисленному типу, достаточно большому для его хранения; требуемый размер зависит от реализации. Функция преобразования также зависит от реализации.

Объект целочисленного типа можно явно преобразовать в указатель. Если целое получено из указателя и имеет достаточно большой размер, это преобразование даст тот же указатель; в противном случае результат зависит от реализации.

Указатель на один тип можно преобразовать в указатель на другой тип. Если исходный указатель ссылается на объект, должным образом не выровненный по границам слов памяти, то в результате может произойти ошибка адресации. Если требования на выравнивание у нового типа меньше или совпадают с требованиями на выравнивание первоначального типа, то гарантируется, что преобразование указателя в другой тип и обратно его не изменит; понятие "выравнивание" зависит от реализации, однако в любой реализации объекты типа char предъявляют минимальные требования на выравнивание. Как описано в А6.8, указатель может также

преобразовываться в void * и обратно, значение указателя при этом не изменяется.

Указатель может быть преобразован в другой указатель того же типа с добавлением или удалением квалификаторов (А4.4, А8.2) того типа объекта, на который этот указатель показывает. Новый указатель, полученный добавлением квалификатора, имеет то же значение, но с дополнительными ограничениями, внесенными новыми квалификаторами. Операция по удалению квалификатора у объекта приводит к тому, что восстанавливается действие его начальных квалификаторов, заданных в объявлении этого объекта.

Наконец, указатель на функцию может быть преобразован в указатель на функцию другого типа. Вызов функции по преобразованному указателю зависит от реализации; однако, если указатель еще раз преобразовать к его исходному типу, результат будет идентичен вызову по первоначальному указателю.

А 6.7. **Тип** void

Значение (несуществующее) объекта типа void никак нельзя использовать, его также нельзя явно или неявно привести к типу, отличному от void. Поскольку выражение типа void обозначает отсутствие значения, его можно применять только там, где не требуется значения. Например, в качестве выражения-инструкции (А9.2) или левого операнда у оператора "запятая" (А7.18).

Выражение можно привести к типу void операцией приведения типа. Например, применительно к вызову функции, используемому в роли выражения-инструкции, операция приведения к void явным образом подчеркивает тот факт, что результат функции отбрасывается.

> Тип void не фигурировал в первом издании этой книги, однако за прошедшее время стал общеупотребительным.

А6.8. Указатели на void

Любой указатель на объект можно привести к типу void * без потери информации. Если результат подвергнуть обратному преобразованию, то мы получим прежний указатель. В отличие от преобразований указатель-в-указатель (рассмотренных в А6.6), которые требуют явных операторов приведения к типу, в присваиваниях и сравнениях указатель любого типа может выступать в паре с указателем типа void * без каких-либо предварительных преобразований типа.

> Такая интерпретация указателей void * - новая; ранее роль обобщенного указателя отводилась указателю типа char *. Стандарт ANSI официально

разрешает использование указателей void * совместно с указателями других типов в присваиваниях и сравнениях; в иных комбинациях указателей стандарт требует явных преобразований типа.

А 7. Выражения

Приоритеты описываемых операторов имеют тот же порядок, что и пункты данного параграфа (от высших к низшим). Например, для оператора +, Описанного в А7.7, термин "операнды" означает "выражения, определенные в А7.1 – А7.6". В каждом пункте описываются операторы, имеющие одинаковый приоритет, и указывается их ассоциативность (левая или правая). Приоритеты и ассоциативность всех операторов отражены в грамматике, приведенной в А13.

Приоритеты и ассоциативность полностью определены, а вот порядок вычисления выражения не определен за некоторым исключением даже для подвыражений с побочным эффектом. Это значит, что если в определении оператора последовательность вычисления его операндов специально не оговаривается, то в реализации можно свободно выбирать любой порядок вычислений и даже чередовать правый и левый порядок. Однако любой оператор использует значения своих операндов в точном соответствии с грамматическим разбором выражения, в котором он встречается.

Это правило отменяет ранее предоставлявшуюся свободу в выборе порядка выполнения операций, которые математически коммутативны и ассоциативны, но которые в процессе вычислений могут таковыми не оказаться. Это изменение затрагивает только вычисления с плавающей точкой, выполняющиеся "на грани точности", и ситуации, когда возможно переполнение.

В языке не определен контроль за переполнением, делением на нуль и другими исключительными ситуациями, возникающими при вычислении выражения. В большинстве существующих реализаций Си при вычислении знаковых целочисленных выражений и присваивании переполнение игнорируется, но результат таких вычислений не определен. Трактовки деления на нуль и всех исключительных ситуаций, связанных с плавающей точкой, могут не совпадать в разных реализациях; иногда для обработки исключительных ситуаций предоставляется нестандартная библиотечная функция.

А 7.1. Генерация указателя

Если тип выражения или подвыражения есть "массив из *T*", где Т - некоторый тип, то значением этого выражения является указатель на первый элемент массива, и тип такого выражения заменяется на тип "указа-

тель на Т". Такая замена типа не делается, если выражение является операндом унарного оператора &, или операндом операций ++, --, sizeof, или левым операндом присваивания, или операндом оператора . (точка). Аналогично, выражение типа "функция, возвращающая *T"*, кроме случая, когда оно является операндом для &, преобразуется в тип "указатель на функцию, возвращающую *T*".

А 7.2. Первичные выражения

Первичные выражения - это идентификаторы, константы, строки и выражения в скобках.

первичное-выражение:
идентификатор
константа
строка
(выражение)

Идентификатор, если он был должным образом объявлен (о том, как это делается, речь пойдет ниже), - первичное выражение. Тип идентификатора специфицируется в его объявлении. Идентификатор есть lvalue, если он обозначает объект (А5) арифметического типа либо объект типа "структура", "объединение" или "указатель".

Константа - первичное выражение. Ее тип зависит от формы записи, которая была рассмотрена в А2.5.

Строковый литерал - первичное выражение. Изначально его тип - "массив из char" ("массив из wchar_t" для строки символов расширенного набора), но в соответствии с правилом, приведенным в А7.1, указанный тип обычно превращается в "указатель на char" ("указатель на wchar_t") с результирующим значением "указатель на первый символ строки". Для некоторых инициализаторов такая замена типа не делается. (См. А8.7.)

Выражение в скобках - первичное выражение, тип и значение которого идентичны типу и значению этого же выражения без скобок. Наличие или отсутствие скобок не влияет на то, является ли данное выражение lvalue или нет.

А 7.3. Постфиксные выражения

В постфиксных выражениях операторы выполняются слева направо.

постфиксное-выражение:
первичное-выражение
постфиксное-выражение [*выражение*]
постфиксное- выражение (*список-аргументов-выражений*$_{необ}$)
постфиксное-выражение . *идентификатор*
постфиксное-выражение -> *идентификатор*

постфиксное-выражение ++
постфиксное-выражение —

список-аргументов-выражений:
выражение-присваивание
список-аргументов-выражений , выражение-присваивание

А7.3.1. Обращение к элементам массива

Постфиксное выражение, за которым следует выражение в квадратных скобках, есть постфиксное выражение, обозначающее обращение к индексируемому массиву. Одно из этих двух выражений должно принадлежать типу "указатель на *T*", где Г- некоторый тип, а другое - целочисленному типу; тип результата индексирования есть *T*. Выражение E1 [E2] по определению идентично выражению `*((E1)+(E2))`. Подробности см. в А8.6.2.

А7.3.2. Вызов функции

Вызов функции есть постфиксное выражение (оно называется именующим выражением функции - *function designator),* за которым следуют скобки, содержащие (возможно пустой) список разделенных запятыми выражений-присваиваний (А7.17), представляющих собой аргументы этой функции. Если постфиксное выражение - идентификатор, не объявленный в текущей области видимости, то считается, что этот идентификатор как бы неявно описан объявлением

extern int *идентификатор*();

помещенным в самом внутреннем блоке, содержащем вызов соответствующей функции. Постфиксное выражение (после, возможно неявного, описания и генерации указателя, см. А7.1) должно иметь тип "указатель на функцию, возвращающую *T*", где *T* – тип возвращаемого значения.

> В первой версии языка для именующего выражения функции допускался только тип "функция", и чтобы вызвать функцию через указатель, требовался явный оператор `*`. ANSI-стандарт поощряет практику некоторых существующих компиляторов, разрешающих иметь одинаковый синтаксис для обращения просто к функции и обращения к функции, специфицированной указателем. Возможность применения старого синтаксиса остается.

Термин *аргумент* используется для выражения, задаваемого в вызове функции; термин *параметр* - для обозначения получаемого ею объекта (или его идентификатора) в определении или объявлении функции. Вместо этих понятий иногда встречаются термины "фактический аргумент (параметр)" и "формальный аргумент (параметр)", имеющие те же смысловыеразличия.

При вызове функции каждый ее аргумент копируется; передача аргументов осуществляется строго через их значения. Функции разрешается изменять значения своих параметров, которые являются лишь копиями аргументов-выражений, но эти изменения не могут повлиять на значения самих аргументов. Однако можно передать указатель, чтобы позволить функции изменить значение объекта, на который указывает этот указатель.

Имеются два способа объявления функции. В новом способе типы параметров задаются явно и являются частью типа функции; такое объявление называется прототипом функции. При старом способе типы параметров не указываются. Способы объявления функций обсуждаются в А8.6.3 и А10.1.

Если вызов находится в области видимости объявления, написанного по-старому, каждый его аргумент подвергается операции повышения типа: для целочисленных аргументов осуществляется целочисленное повышение (А6.1), а для аргументов типа `float` - преобразование в `double`. Если число аргументов не соответствует количеству параметров в определении функции или если типы аргументов после повышения не согласуются с типами соответствующих параметров, результат вызова не определен. Критерий согласованности типов зависит от способа определения функции (старого или нового). При старом способе сравниваются повышенный тип аргумента в вызове и повышенный тип соответствующего параметра; при новом способе повышенный тип аргумента и тип параметра (без его повышения) должны быть одинаковыми.

Если вызов находится в области видимости объявления, написанного по-новому, аргументы преобразуются, как если бы они присваивались переменным, имеющим типы соответствующих параметров прототипа. Число аргументов должно совпадать с числом явно описанных параметров, если только список параметров не заканчивается многоточием (`, ...`). В противном случае число аргументов должно быть больше числа параметров или равно ему; "скрывающиеся" под многоточием аргументы подвергаются операции повышения типа (так, как это было описано в предыдущем абзаце). Если определение функции задано по-старому, то типы параметров в прототипе, которые неявно присутствуют в вызове, должны соответствовать типам параметров в определении функции после их повышения.

Эти правила особенно усложнились из-за того, что они призваны обслуживать смешанный способ (старого с новым) задания функций. По возможности его следует избегать.

Очередность вычисления аргументов не определяется, в разных компиляторах она различна. Однако гарантируется, что аргументы и именующее

9 Зак. 1116

выражение функции вычисляются полностью (включая и побочные эффекты) до входа в нее. Любая функция допускает рекурсивное обращение.

А7.3.3. Обращение к структурам

Постфиксное выражение, за которым стоит точка с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть структурой или объединением, а идентификатор - именем элемента структуры или объединения. Значение - именованный элемент структуры или объединения, а тип значения - тип элемента структуры или объединения. Выражение является lvalue, если первое выражение - lvalue и если тип второго выражения - не "массив".

Постфиксное выражение, за которым стоит стрелка (составленная из знаков - и >) с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть указателем на структуру (объединение), а идентификатор - именем элемента структуры (объединения). Результат - именованный элемент структуры (объединения), на которую указывает указатель, а тип значения — тип элемента структуры (объединения); результат - lvalue, если тип не есть "массив".

Таким образом, выражение `E1->MOS` означает то же самое, что и выражение `(*E1).MOS`. Структуры и объединения рассматриваются в А8.3.

> В первом издании книги уже было приведено правило, по которому имя элемента должно принадлежать структуре или объединению, упомянутому в постфиксном выражении. Там, однако, оговаривалось, что оно не является строго обязательным. Последние компиляторы и ANSI делают его обязательным.

А7.3.4. Постфиксные операторы инкремента и декремента

Постфиксное выражение, за которым следует ++ или --, есть постфиксное выражение. Значением такого выражения является значение его операнда. После того как значение было взято, операнд увеличивается (++) или уменьшается (--) на 1. Операнд должен быть lvalue; информация об ограничениях, накладываемых на операнд, и деталях операций содержится в А7.7, где обсуждаются аддитивные операторы, и в А7.17, где рассматривается присваивание. Результат инкрементирования или декрементирования не есть lvalue.

А7.4. Унарные операторы

Выражения с унарными операторами выполняются справа налево.

унарное-выражение:
    постфиксное-выражение

++ *унарное-выражение*
— *унарное-выражение*
унарный-оператор выражение-приведенное-к-типу
sizeof *унарное-выражение*
sizeof (*имя-типа*)

унарный-оператор: один из
& * + ~ !

А7.4.1. Префиксные операторы инкремента и декремента

Унарное выражение, перед которым стоит ++ или --, есть унарное выражение. Операнд увеличивается (++) или уменьшается (--) на 1.

Значением выражения является значение его операнда после увеличения (уменьшения). Операнд всегда должен быть lvalue; информация об ограничениях на операнд и о деталях операции содержится в А7.7, где обсуждаются аддитивные операторы, и в А7.17, где рассматривается присваивание. Результат инкрементирования и декрементирования не есть lvalue.

А7.4.2. Оператор получения адреса

Унарный оператор & обозначает операцию получения адреса своего операнда. Операнд должен быть либо lvalue, не ссылающимся ни на битовое поле, ни на объект, объявленный как register, либо иметь тип "функция". Результат - указатель на объект (или функцию), адресуемый этим lvalue. Если тип операнда есть *Т,* то типом результата является "указатель на *Т"*.

А7.4.3. Оператор косвенного доступа

Унарный оператор * обозначает операцию косвенного доступа (раскрытия указателя), возвращающую объект (или функцию), на который указывает ее операнд. Результат есть lvalue, если операнд - указатель на объект арифметического типа или на объект типа "структура", "объединение" или "указатель". Если тип выражения - "указатель на *Т"*, то тип результата - *Т*.

А7.4.4. Оператор унарный плюс

Операнд унарного + должен иметь арифметический тип, результат - значение операнда. Целочисленный операнд подвергается целочисленному повышению. Типом результата является повышенный тип операнда.

Унарный + был добавлен для симметрии с унарным -.

А7.4.5. Оператор унарный минус

Операнд для унарного - должен иметь арифметический тип, результат - значение операнда с противоположным знаком. Целочисленный

операнд подвергается целочисленному повышению. Отрицательное значение от беззнаковой величины вычисляется вычитанием из nmax+1 приведенного к повышенному типу операнда, где nmax - максимальное число повышенного типа; однако минус нуль есть нуль. Типом результата будет повышенный тип операнда.

А 7.4.6. Оператор побитового отрицания

Операнд оператора ~ должен иметь целочисленный тип, результат - дополнение операнда до единиц по всем разрядам. Выполняется целочисленное повышение типа операнда. Если операнд беззнаковый, то результат получается вычитанием его значения из самого большого числа повышенного типа. Если операнд знаковый, то результат вычисляется посредством приведения "повышенного операнда" к беззнаковому типу, выполнения операции ~ и обратного приведения его к знаковому типу. Тип результата - повышенный тип операнда.

А 7.4.7. Оператор логического отрицания

Операнд оператора ! должен иметь арифметический тип или быть указателем. Результат равен 1, если сравнение операнда с 0 дает истину, и равен 0 в противном случае. Тип результата - int.

А 7.4.8. Оператор определения размера sizeof

Оператор sizeof дает число байтов, требуемое для хранения объекта того типа, который имеет его операнд. Операнд - либо выражение (которое не вычисляется), либо имя типа, записанное в скобках. Примененный к char оператор sizeof дает 1. Для массива результат равняется общему количеству байтов в массиве, для структуры или объединения - числу байтов в объекте, включая и байты-заполнители, которые понадобились бы, если бы из элементов составлялся массив. Размер массива из n элементов всегда равняется n, помноженному на размер отдельного его элемента. Данный оператор нельзя применять к операнду типа "функция", к незавершенному типу и к битовому полю. Результат - беззнаковая целочисленная константа; конкретный ее тип зависит от реализации. В стандартном заголовочном файле <stddef.h> (см. приложение B) этот тип определяется под именем size_t.

А 7.5. Оператор приведения типа

Имя типа, записанное перед унарным выражением в скобках, вызывает приведение значения этого выражения к указанному типу.

выражение-приведенное-к-типу:
  унарное-выражение
  (имя-типа) выражение-приведенное-к-типу

Данная конструкция называется *приведением*. Имена типов даны в А8.8. Результат преобразований описан в А6. Выражение с приведением типа не является lvalue.

А 7.6. Мультипликативные операторы

Мультипликативные операторы *, / и % выполняются слева направо.

мультипликативное-выражение:
выражение-приведенное-к-типу
мулыпипликативное-выражение * *выражение-приведенное-к-типу*
мулыпипликативное-выражение / *выражение-приведенное-к-типу*
мультипликативное-выражение % *выражение-приведенное-к-типу*

Операнды операторов * и / должны быть арифметического типа, оператора % - целочисленного типа. Над операндами осуществляются обычные арифметические преобразования, которые приводят их значения к типу результата.

Бинарный оператор * обозначает умножение.

Бинарный оператор / получает частное, а % - остаток от деления первого операнда на второй; если второй операнд есть 0, то результат не определен. В противном случае всегда выполняется соотношение: `(a/b)*b` + a%b равняется a. Если оба операнда не отрицательные, то остаток не отрицательный и меньше делителя; в противном случае стандарт гарантирует только одно: что абсолютное значение остатка меньше абсолютного значения делителя.

А 7.7. Аддитивные операторы

Аддитивные операторы + и - выполняются слева направо. Если операнды имеют арифметический тип, то осуществляются обычные арифметические преобразования. Для каждого оператора существует еще несколько дополнительных сочетаний типов.

аддитивное-выражение:
мультипликативное -выражение
аддитивное-выражение + *мультипликативное-выражение*
аддитивное-выражение - *мулыпипликативное-выражение*

Результат выполнения оператора + есть сумма его операндов. Указатель на объект в массиве можно складывать с целочисленным значением. При этом последнее преобразуется в адресное смещение посредством умножения его на размер объекта, на который ссылается указатель. Сумма является указателем на объект того же типа; только ссылается этот указатель на другой объект того же массива, отстоящий от первоначального соответственно вычисленному смещению. Так, если P - указатель на объект

в массиве, то P+1 - указатель на следующий объект того же массива. Если полученный в результате суммирования указатель указывает за границы массива, то, кроме случая, когда он указывает на место, находящееся непосредственно за концом массива, результат будет неопределенным.

Возможность для указателя указывать на элемент, расположенный сразу за концом массива, является новой. Тем самым узаконена общепринятая практика организации циклического перебора элементов массива.

Результат выполнения оператора - (минус) есть разность операндов. Из указателя можно вычитать значение любого целочисленного типа с теми же преобразованиями и при тех же условиях, что и в сложении.

Если к двум указателям на объекты одного и того же типа применить оператор вычитания, то в результате получится целочисленное значение со знаком, представляющее собой расстояние между объектами, на которые указывают эти указатели; указатель на следующий объект на 1 больше указателя на предыдущий объект. Тип результата зависит от реализации; в стандартном заголовочном файле <stddef.h> он определен под именем ptrdiff_t. Значение не определено, если указатели указывают на объекты не одного и того же массива; однако если P указывает на последний элемент массива, то P+1-P имеет значение, равное 1.

А 7.8. Операторы сдвига

Операторы сдвига << и >> выполняются слева направо. Для обоих операторов каждый операнд должен иметь целочисленный тип, и каждый из них подвергается целочисленному повышению. Тип результата совпадает с повышенным типом левого операнда. Результат не определен, если правый операнд отрицателен или его значение превышает число битов в типе левого выражения или равно ему.

сдвиговое-выражение:
аддитивное-выражение
сдвиговое-выражение >> *аддитивное-выражение*
сдвиговое-выражение << *аддитивное-выражение*

Значение E1<<E2 равно значению E1 (рассматриваемому как цепочка битов), сдвинутому влево на E2 битов; при отсутствии переполнения такая операция эквивалентна умножению на 2^{E2}. Значение E1>>E2 равно значению E1, сдвинутому вправо на E2 битовые позиции. Если E1 - беззнаковое или имеет неотрицательное значение, то правый сдвиг эквивалентен делению на 2^{E2}, в противном случае результат зависит от реализации.

А 7.9. Операторы отношения

Операторы отношения выполняются слева направо, однако это свойство едва ли может оказаться полезным; согласно грамматике языка вы-

ражение `a<b<c` трактуется так же, как `(a<b)<c`, а результат вычисления `a<b` может быть только 0 или 1.

выражение-отношения:
сдвиговое-выражение
выражение-отношения < *сдвиговое-выражениевыражение-отношения* >
сдвиговое-выражениевыражение-отношения <= *сдвиговое-выражение*
выражение-отношения >= *сдвиговое-выражение*

Операторы: < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) — все выдают 0, если специфицируемое отношение ложно, и 1, если оно истинно. Тип результата - `int`. Над арифметическими операндами выполняются обычные арифметические преобразования. Можно сравнивать указатели на объекты одного и того же типа (без учета квалификаторов); результат будет зависеть от относительного расположения в памяти. Допускается, однако, сравнение указателей на разные части одного и того же объекта: если два указателя указывают на один и тот же простой объект, то они равны; если они указывают на элементы одной структуры, то указатель на элемент с более поздним объявлением в структуре больше; если указатели указывают на элементы одного и того же объединения, то они равны; если указатели указывают на элементы некоторого массива, то сравнение этих указателей эквивалентно сравнению их индексов. Если P указывает на последний элемент массива, то `P+1` больше, чем P, хотя `P+1` указывает за границы массива. В остальных случаях результат сравнения не определен.

Эти правила несколько ослабили ограничения, установленные в первой редакции языка. Они позволяют сравнивать указатели на различные элементы структуры и объединения и легализуют сравнение с указателем на место, которое расположено непосредственно за концом массива.

А 7.10. Операторы равенства

выражение-равенства:
выражение-отношения
выражение-равенства == *выражение-отношения*
выражение-равенства != *выражение-отношения*

Операторы == (равно) и != (не равно) аналогичны операторам отношения с той лишь разницей, что имеют более низкий приоритет. (Таким образом, `a<b == c<d` есть 1 тогда и только тогда, когда отношения `a<b` и `c<d` или оба истинны, или оба ложны.)

Операторы равенства подчиняются тем же правилам, что и операторы отношения. Кроме того, они дают возможность сравнивать указатель с целочисленным константным выражением, значение которого равно нулю, и с указателем на `void` (см. А6.6.).

A 7.11. Оператор побитового *И*

И-выражение:
выражение-равенства
И-выражение & *выражение-равенства*

Выполняются обычные арифметические преобразования; результат - побитовое *И* операндов. Оператор применяется только к целочисленным операндам.

A 7.12. Оператор побитового исключающего *ИЛИ*

исключающее-ИЛИ-выражение:
И-выражение
исключающее-ИЛИ-выражение ^ *И-выражение*

Выполняются обычные арифметические преобразования; результат - побитовое исключающее *ИЛИ* операндов. Оператор применяется только к целочисленным операндам.

A 7.13. Оператор побитового *ИЛИ*

ИЛИ-выражение:
исключающее-ИЛИ-выражение
ИЛИ-выражение ! *исключающее-ИЛИ-выражение*

Выполняются обычные арифметические преобразования; результат – побитовое *ИЛИ* операндов. Оператор применяется только к целочисленным операндам.

A 7.14. Оператор логического *И*

логическое-И-выражение:
ИЛИ-выражение
логическое-И-выражение && *ИЛИ-выражение*

Операторы && выполняются слева направо. Оператор && выдает 1, если оба операнда не равны нулю, и 0 в противном случае. В отличие от &, && гарантирует, что вычисления будут проводиться слева направо: вычисляется первый операнд со всеми побочными эффектами; если он равен О, то значение выражения есть 0. В противном случае вычисляется правый операнд, и, если он равен 0, то значение выражения есть 0, в противном случае оно равно 1.

Операнды могут принадлежать к разным типам, но при этом каждый из них должен иметь либо арифметический тип, либо быть указателем. Тип результата – int.

А 7.15. Оператор логического *ИЛИ*

логическое-ИЛИ-выражение:
логическое-И-выражение
логическое-ИЛИ-выражение || *логическое-И-выражение*

Операторы || выполняются слева направо. Оператор || выдает 1, если по крайней мере один из операндов не равен нулю, и 0 в противном случае. В отличие от |, оператор || гарантирует, что вычисления будут проводиться слева направо: вычисляется первый операнд, включая все побочные эффекты; если он не равен 0, то значение выражения есть 1. В противном случае вычисляется правый операнд, и если он не равен 0, то значение выражения есть 1, в противном случае оно равно 0.

Операнды могут принадлежать разным типам, но операнд должен иметь либо арифметический тип, либо быть указателем. Тип результата - int.

А 7.16. Условный оператор

условное-выражение:
логическое-ИЛИ-выражение
логическое-ИЛИ-выражение ? *выражение* : *условное-выражение*

Вычисляется первое выражение, включая все побочные эффекты; если оно не равно 0, то результат есть значение второго выражения, в противном случае - значение третьего выражения. Вычисляется только один из двух последних операндов: второй или третий. Если второй и третий операнды арифметические, то выполняются обычные арифметические преобразования, приводящие к некоторому общему типу, который и будет типом результата. Если оба операнда имеют тип void, или являются структурами или объединениями одного и того же типа, или представляют собой указатели на объекты одного и того же типа, то результат будет иметь тот же тип, что и операнды. Если один из операндов имеет тип "указатель", а другой является константой 0, то 0 приводится к типу "указатель", этот же тип будет иметь и результат. Если один операнд является указателем на void, а второй - указателем другого типа, то последний преобразуется в указатель на void, который и будет типом результата.

При сравнении типов указателей квалификаторы типов (А8.2) объектов, на которые указатели ссылаются, во внимание не принимаются, но тип результата наследует квалификаторы обеих ветвей условного выражения.

А 7.17. Выражения присваивания

Существует несколько операторов присваивания; они выполняются справа налево.

выражение-присваивания:
условное-выражение
унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из

= *= /= %= += -= <<= >>= &= ^= |=

Операторы присваивания в качестве левого операнда требуют lvalue, причем модифицируемого; это значит, что оно не может быть массивом, или иметь незавершенный тип, или быть функцией. Тип левого операнда, кроме того, не может иметь квалификатора const; и, если он является структурой или объединением, в них не должно быть элементов или подэлементов (для вложенных структур или объединений) с квалификаторами const.

Тип выражения присваивания соответствует типу его левого операнда, а значение равно значению его левого операнда после завершения присваивания.

В простом присваивании с оператором = значение выражения замещает объект, к которому обращается lvalue. При этом должно выполняться одно из следующих условий: оба операнда имеют арифметический тип (если типы операндов разные, правый операнд приводится к типу левого операнда); оба операнда есть структуры или объединения одного и того же типа; один операнд есть указатель, а другой - указатель на void; левый операнд - указатель, а правый - константное выражение со значением 0; оба операнда - указатели на функции или объекты, имеющие одинаковый тип (за исключением возможного отсутствия const или volatile у правого операнда).

Выражение E1 *op* = E2 эквивалентно выражению E1 = E1 *op* (E2) с одним исключением: E1 вычисляется только один раз.

А7.18. Оператор запятая

выражение:
выражение-присваивания
выражение , выражение-присваивания

Два выражения, разделенные запятой, вычисляются слева направо, и значение левого выражения отбрасывается. Тип и значение результата совпадают с типом и значением правого операнда. Вычисление всех побочных эффектов левого операнда завершается перед началом вычисления правого операнда. В контексте, в котором запятая имеет специальное значение, например в списках аргументов функций (А7.3.2) или в списках инициализаторов (А8.7) (здесь в качестве синтаксических единиц фигу-

рируют выражения присваивания), оператор запятая может появиться только в группирующих скобках. Например, в

```
f(a, (t=3, t+2), c)
```

три аргумента, из которых второй имеет значение 5.

А 7.19. Константные выражения

Синтаксически, константное выражение - это выражение с ограниченным подмножеством операторов:

константное-выражение:
  условное -выражение

При указании case-меток в переключателе, задании границ массивов и длин полей битов, на месте значений перечислимых констант и инициализаторов, а также в некоторых выражениях для препроцессора требуются выражения, вычисление которых приводит к константе.

Константные выражения не могут содержать присваиваний, операторов инкрементирования и декрементирования, вызовов функций и операторов-запятых; перечисленные ограничения не распространяются на операнд оператора `sizeof`. Если требуется получить целочисленное константное выражение, то его операнды должны состоять из целых, перечислимых (`enum`), символьных констант и констант с плавающей точкой; операции приведения должны специфицировать целочисленный тип, а любая константа с плавающей точкой - приводиться к целому. Из этого следует, что в константном выражении не может быть массивов, операций косвенного обращения (раскрытия указателя), получения адреса и доступа к полям структуры. (Однако для sizeof возможны операнды любого вида.)

Для константных выражений в инициализаторах допускается большая свобода; операндами могут быть константы любого типа, а к внешним или статическим объектам и внешним и статическим массивам, индексируемым константными выражениями, возможно применять унарный оператор &. Унарный оператор & может также неявно присутствовать при использовании массива без индекса или функции без списка аргументов. Вычисление инициализатора должно давать константу или адрес ранее объявленного внешнего или статического объекта плюс-минус константа.

Меньшая свобода допускается для целочисленных константных выражений, используемых после `#if`: не разрешаются `sizeof`-выражения, константы типа `enum` и операции приведения типа. (См. А12.5.)

А8. Объявления

То, каким образом интерпретируется каждый идентификатор, специфицируется объявлениями; они не всегда резервируют память для описываемых ими идентификаторов. Объявления, резервирующие память, называются *определениями* и имеют следующий вид:

объявление:
  спецификаторы-объявления список-инициализаторов-объявителей$_{\text{необ}}$

Объявители в списке-инициализаторов-объявителей содержат объявляемые идентификаторы; спецификаторы-объявления представляют собой последовательности, состоящие из спецификаторов типа и класса памяти.

спецификаторы-объявления:
  спецификатор-класса-памяти спецификаторы-объявления$_{\text{необ.}}$
  спецификатор-типа спецификаторы-объявления$_{\text{необ.}}$
  квалификатор-типа спецификаторы-объявления$_{\text{необ.}}$

список-инициализаторов-объявителей:
  инициализатор-объявитель
  список-инициализаторов-объявителей , инициализатор-объявитель

инициализатор-объявитель:
  объявитель
  объявитель = *инициализатор*

Объявители содержат подлежащие объявлению имена. Мы рассмотрим их позже, в А8.5. Объявление должно либо иметь по крайней мере один объявитель, либо его спецификатор типа должен определять тег структуры или объединения, либо - задавать элементы перечисления; пустое объявление недопустимо.

А 8.1. Спецификаторы класса памяти

Класс памяти специфицируется следующим образом:

спецификатор-класса-памяти:
  auto
  register
  static
  extern
  typedef

Смысл классов памяти обсуждался в А4.

Спецификаторы auto и register дают объявляемым объектам класс автоматической памяти, и эти спецификаторы можно применять только

внутри функции. Объявления с auto и register одновременно являются определениями и резервируют память. Спецификатор register эквивалентен auto, но содержит подсказку, сообщающую, что в программе объявленные им объекты используются интенсивно. На регистрах может быть размещено лишь небольшое число объектов, причем определенного типа; указанные ограничения зависят от реализации. В любом случае к register-объекту нельзя применять (явно или неявно) унарный оператор &.

> Новым является правило, согласно которому вычислять адрес объекта класса register нельзя, а класса auto можно.

Спецификатор static дает объявляемым объектам класс статической памяти, он может использоваться и внутри, и вне функций. Внутри функции этот спецификатор вызывает выделение памяти и служит определением; его роль вне функций будет объяснена в А11.2.

Объявление со спецификатором extern, используемое внутри функции, объявляет, что для объявляемого объекта где-то выделена память; о ее роли вне функций будет сказано в А11.2.

Спецификатор typedef не резервирует никакой памяти и назван спецификатором класса памяти из соображений стандартности синтаксиса; речь об этом спецификаторе пойдет в А8.9.

Объявление может содержать не более одного спецификатора класса памяти. Если он в объявлении отсутствует, то действуют следующие правила: считается, что объекты, объявляемые внутри функций, имеют класс auto; функции, объявляемые внутри функций, - класс extern; объекты и функции, объявляемые вне функций, - статические и имеют внешние связи (см. А10, АИ).

А 8.2. Спецификаторы типа

Спецификаторы типа определяются следующим образом:

спецификатор-типа:
- void
- char
- short
- int
- long
- float
- double
- signed
- unsigned
- *структуры-или-объединения-спецификатор*
- *спецификатор-перечисления*
- *typedef-имя*

Вместе с int допускается использование еще какого-то одного слова - long или short; причем сочетание long int имеет тот же смысл, что и просто long; аналогично short int - то же самое, что и short. Слово long может употребляться вместе с double. С int и другими его модификациями (short, long или char) разрешается употреблять одно из слов signed или unsigned. Любое из последних может использоваться самостоятельно, в этом случае подразумевается int.

Спецификатор signed бывает полезен, когда требуется обеспечить, чтобы объекты типа char имели знак; его можно применять и к другим целочисленным типам, но в этих случаях он избыточен.

За исключением описанных выше случаев объявление не может содержать более одного спецификатора типа. Если в объявлении нет ни одного спецификатора типа, то имеется в виду тип int.

Для указания особых свойств объявляемых объектов предназначаются квалификаторы:

квалификатор-типа:
  const
  volatile

Квалификаторы типа могут употребляться с любым спецификатором типа. Разрешается инициализировать const-объект, однако присваивать ему что-либо в дальнейшем запрещается. Смысл квалификатора volatile зависит от реализации.

> Средства const и volatile (изменчивый) введены стандартом ANSI. Квалификатор const применяется, чтобы разместить объекты в памяти, открытой только на чтение (ПЗУ), или чтобы способствовать возможной оптимизации. Назначение квалификатора volatile - подавить оптимизацию, которая без этого указания могла бы быть проведена. Например, в машинах, где адреса регистров ввода-вывода отображены на адресное пространство памяти, указатель на регистр некоторого устройства мог бы быть объявлен как volatile, чтобы запретить компилятору экономить очевидно избыточную ссылку через указатель. Компилятор может игнорировать указанные квалификаторы, однако обязан сигнализировать о явных попытках изменить значение const-объектов.

А 8.3. Объявления структур и объединений

Структура - это объект, состоящий из последовательности именованных элементов различных типов. Объединение - объект, который в каждый момент времени содержит один из нескольких элементов различных типов. Объявления структур и объединений имеют один и тот же вид.

спецификатор структуры-или-объединения:
  структуры-или-объединенияидентификатор$_{необ}$ { список-объявлений-структуры}

структуры-или-объединения идентификатор

структура-или-объединение:

struct

union

Список-объявлений-структуры является последовательностью объявлений элементов структуры или объединения:

список-объявлений-структуры:
объявление-структуры
список-объявлений-структуры объявление-структуры

объявление-структуры:
список-спецификаторов-квалификаторов список-структуры-объявителей;

список-спецификаторов-квалификаторов:
спецификатор-типа список-спецификаторов-квалификаторов$_{необ}$
квалификатор-типа список-спецификаторов-квалификаторов$_{необ}$

список-структуры-объявителей:
структуры-объявитель
список-структуры-объявителей , структуры-объявитель

Обычно объявление-структуры является просто объявлением для элементов структуры или объединения. Элементы структуры, в свою очередь, могут состоять из заданного числа разрядов (битов). Такой элемент называется *битовым полем* или просто полем. Его размер отделяется от имени поля двоеточием:

структуры-объявитель:
объявитель
объявитель$_{необ}$: *константное-выражение*

Спецификатор типа, имеющий вид

структуры-или-объединения идентификатор { *список-объявлений-структуры* }

объявляет идентификатор *тегом* структуры или объединения, специфицированных списком. Последующее объявление в той же или внутренней области видимости может обращаться к тому же типу, используя в спецификаторе тег без списка:

структуры-или-объединения идентификатор

Если спецификатор с тегом, но без списка появляется там, где тег не объявлен, специфицируется *незавершенный тип.* Объекты с незавершенным типом структуры или объединения могут упоминаться в контексте,

где не требуется знать их размер — например в объявлениях (но не определениях) для описания указателя или создания typedef, но не в иных случаях. Тип становится завершенным при появлении последующего спецификатора с этим тегом, содержащего список объявлений. Даже в спецификаторах со списком объявляемый тип структуры или объединения является незавершенным внутри списка и становится завершенным только после появления символа }, заканчивающего спецификатор.

Структура не может содержать элементов незавершенного типа. Следовательно, невозможно объявить структуру или объединение, которые содержат сами себя. Однако, кроме придания имени типу структуры или объединения, тег позволяет определять структуры, обращающиеся сами к себе; структура или объединение могут содержать указатели на самих себя, поскольку указатели на незавершенные типы объявлять можно.

Особое правило применяется к объявлениям вида

структуры-или-объединенияидентификатор ;

которые объявляют структуру или объединение, но не имеют списка объявления и объявителя. Даже если идентификатор имеет тег структуры или объединения во внешней области видимости (A11.1), это объявление делает идентификатор тегом новой структуры или объединения незавершенного типа во внутренней области видимости.

> Это невразумительное правило - новое в ANSI. Оно предназначено для взаимно рекурсивных структур, объявленных во внутренней области видимости, но теги которых могут быть уже объявлены во внешней области видимости.

Спецификатор структуры или объединения со списком, но без тега создает уникальный тип, к которому можно обращаться непосредственно только в объявлении, частью которого он является.

Имена элементов и тегов не конфликтуют друг с другом или обычными переменными. Имя элемента не может появляться дважды в одной и той же структуре или объединении, но тот же элемент можно использовать в разных структурах или объединениях.

> В первой редакции этой книги имена элементов структуры и объединения не связывались со своими родителями. Однако в компиляторах эта связь стала обычной задолго до появления стандарта ANSI.

Элемент структуры или объединения, не являющийся полем, может иметь любой тип объекта. Поле (которое не имеет объявителя и, следовательно, может быть безымянным) имеет тип int, unsigned int или signed int и интерпретируется как объект целочисленного типа указанной в битах длины. Считается ли поле int знаковым или беззнаковым, зависит отреализации. Соседнийэлемент-полеупаковываетсявячейкипамяти

в зависимости от реализации в зависящем от реализации направлении. Когда следующее за полем другое поле не влезает в частично заполненную ячейку памяти, оно может оказаться разделенным между двумя ячейками, или ячейка может быть забита балластом. Безымянное поле нулевой ширины обязательно приводит к такой забивке, так что следующее поле начнется с края следующей ячейки памяти.

Стандарт ANSI делает поля еще более зависимыми от реализации, чем в первой редакции книги. Чтобы хранить битовые поля в "зависящем от реализации" виде без квалификации, желательно прочитать правила языка. Структуры с битовыми полями могут служить переносимым способом для попытки уменьшить размеры памяти под структуру (вероятно, ценой увеличения кода программы и времени на доступ к полям) или непереносимым способом для описания распределения памяти на битовом уровне. Во втором случае необходимо понимать правила местной реализации.

Элементы структуры имеют возрастающие по мере объявления элементов адреса. Элементы структуры, не являющиеся полями, выравниваются по границам адресов в зависимости от своего типа; таким образом, в структуре могут быть безымянные дыры. Если указатель на структуру приводится к типу указателя на ее первый элемент, результат указывает на первый элемент.

Объединение можно представить себе как структуру, все элементы которой начинаются со смещением 0 и размеры которой достаточны для хранения любого из элементов. В любой момент времени в объединении хранится не больше одного элемента. Если указатель на объединение приводится к типу указателя на один из элементов, результат указывает на этот элемент.

Вот простой пример объявления структуры:

```
struct tnode {
    char tword[20];
    int  count;
    struct tnode *left;
    struct tnode *right;
};
```

Эта структура содержит массив из 20 символов, число типа int и два указателя на подобную структуру. Если дано такое объявление, то

```
struct tnode s, *sp;
```

объявит s как структуру заданного вида, а sp – как указатель на такую структуру. Согласно приведенным определениям выражение

```
sp->count
```

обращается к элементу count в структуре, на которую указывает sp;

```
s.left
```

- указатель на левое поддерево в структуре s; а

```
s.right->tword[0]
```

- это первый символ из tword - элемента правого поддерева s.

Вообще говоря, невозможно проконтролировать, тот ли используется элемент объединения, которому последний раз присваивалось значение. Однако гарантируется выполнение правила, облегчающего работу с элементами объединения: если объединение содержит несколько структур, начинающихся с общей для них последовательности данных, и если объединение в текущий момент содержит одну из этих структур, то к общей части данных разрешается обращаться через любую из указанных структур. Так, правомерен следующий фрагмент программы:

```
union {
   struct {
      int  type;
   } n;
   struct {
      int  type;
      int  intnode;
   }  ni;
   struct {
      int  type;
      float floatnode;
   } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if  (u.n.type == FLOAT)
... sin(u.nf.floatnode) ...
```

А8.4. Перечисления

Перечисления - это уникальный тип, значения которого покрываются множеством именованных констант, называемых перечислителями. Вид спецификатора перечисления заимствован у структур и объединений.

спецификатор-перечисления:
   enum *идентификатор*$_{необ}$ { *список-перечислителей* }
   enum *идентификатор*

список-перечислителей:
перечислитель
список-перечислителей , перечислитель
перечислитель:
идентификатор
идентификатор = константное-выражение

Идентификаторы, входящие в список перечислителей, объявляются константами типа `int` и могут употребляться везде, где требуется константа. Если в этом списке нет ни одного перечислителя со знаком =, то значения констант начинаются с 0 и увеличиваются на 1 по мере чтения объявления слева направо. Перечислитель со знаком = даёт соответствующему идентификатору значение; последующие идентификаторы продолжают прогрессию от заданного значения.

Имена перечислителей, используемые в одной области видимости, должны отличаться друг от друга и от имен обычных переменных, однако их значения могут и совпадать.

Роль идентификатора в переч-спецификаторе аналогична роли тега структуры в структ-спецификаторе: он является именем некоторого конкретного перечисления. Правила для списков и переч-спецификаторов (с тегами и без) те же, что и для спецификаторов структур или объединений, с той лишь оговоркой, что элементы перечислений не бывают незавершенного типа; тег переч-спецификатора без списка перечислителей должен иметь в пределах области видимости спецификатор со списком.

В первой версии языка перечислений не было, но они уже несколько лет применяются.

А 8.5. Объявители

Объявители имеют следующий синтаксис:

объявитель:
указатель$_{необ}$ *собственно-объявитель*
собственно -объявитель:
идентификатор
(*объявитель*)
собственно-объявитель [*константное-выражение*$_{необ}$]
собственно-объявитель (*список-типов-параметров*)
собственно-объявитель (*список-идентификаторов*$_{необ}$)
указатель:
* *список-квалификаторов-типа*$_{необ}$
* *список-квалификаторов-типа*$_{необ}$ *указатель*
список-квалификаторов-типа:

квалификатор-типа
список-квалификаторов-типа квалификатор-типа

У структуры объявителя много сходных черт со структурой подвыражений, поскольку в объявителе, как и в подвыражении, допускаются операции косвенного обращения, обращения к функции и получения элемента массива (с тем же порядком применения).

А 8.6. Что означают объявители

Список объявителей располагается сразу после спецификаторов типа и указателя класса памяти. Главный элемент любого объявителя - это объявляемый им идентификатор; в простейшем случае объявитель из него одного и состоит, что отражено в первой строке продукции грамматики с именем *собственно-объявитель*. Спецификаторы класса памяти относятся непосредственно к идентификатору, а его тип зависит от вида объявителя. Объявитель следует воспринимать как утверждение: если в выражении идентификатор появляется в том же контексте, что и в объявителе, то он обозначает объект специфицируемого типа.

Если соединить спецификаторы объявления, относящиеся к типу (А8.2), и некоторый конкретный объявитель, то объявление примет вид "T D", где T - тип, а D - объявитель. Эта запись индуктивно придает тип идентификатору любого объявителя.

В объявлении T D, где D - просто идентификатор, тип идентификатора есть T.

В объявлении T D, где D имеет вид

```
( D1 )
```

тип идентификатора в D1 тот же, что и в D. Скобки не изменяют тип, но могут повлиять на результаты его "привязки" к идентификаторам в сложных объявителях.

А 8.6.1. Объявители указателей

В объявления T D, где D имеет вид

* *список-квалификаторов-типа*$_{необ}$ D1

а тип идентификатора объявления T D1 есть "*модификатор-типа* T", тип идентификатора D есть "*модификатор-типа список-квалификаторов-типа* указатель на T". Квалификаторы, следующие за *, относятся к самому указателю, а не к объекту, на который он указывает. Рассмотрим, например, объявление

```
int *ap[];
```

Здесь ap[] играет роль D1; объявление int ap[] следует расшифровать (см. ниже) как "массив из int"; список квалификаторов типа здесь пуст, а модификатор типа есть "массив из". Следовательно, на самом деле объявление ap гласит: "массив из указателей на int". Вот еще примеры объявлений:

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

В них объявляются целое i и указатель на целое pi. Значение указателя cpi неизменно; cpi всегда будет указывать в одно и то же место, даже если значение, на которое он указывает, станет иным. Целое ci есть константа, оно измениться не может (хотя может инициализироваться, как в данном случае). Тип указателя pci произносится как "указатель на const int"; сам указатель можно изменить; при этом он будет указывать на другое место, но значение, на которое он будет указывать, с помощью pci изменить нельзя.

А 8.6.2. Объявители массивов

В объявления T D, где D имеет вид

D1 [*константное-выражение*$_{необ}$]

и где тип идентификатора объявления T D1 есть *"модификатор-типа* T", тип идентификатора 0 есть *"модификатор-типа* массив из T". Если константное выражение присутствует, то оно должно быть целочисленным и больше 0. Если константное выражение, специфицирующее количество элементов в массиве, отсутствует, то массив имеет незавершенный тип.

Массив можно конструировать из объектов арифметического типа, указателей, структур и объединений, а также других массивов (генерируя при этом многомерные массивы). Любой тип, из которого конструируется массив, должен быть завершенным, он не может быть, например, структурой или массивом незавершенного типа. Это значит, что для многомерного массива пустой может быть только первая размерность. Незавершенный тип массива получает свое завершение либо в другом объявлении этого массива (А10.2), либо при его инициализации (А8.7). Например, запись

```
float fa[17], *afp[17];
```

объявляет массив из чисел типа float и массив из указателей на числа типа float. Аналогично

```
static int x3d[3][5][7];
```

объявляет статический трехмерный массив целых размера 3x5x7. На самом деле, если быть точными, x3d является массивом из трех элементов,

каждый из которых есть массив из пяти элементов, содержащих по 7 значений типа int.

Операция индексирования E1[E2] определена так, что она идентична операции *(E1+E2). Следовательно, несмотря на асимметричность записи, индексирование - коммутативная операция. Учитывая правила преобразования, применяемые для оператора + и массивов (А6.6, А7.1, А7.7), можно сказать, что если E1 - массив, а E2 - целое, то E1[E2] обозначает E2-й элемент массива E1.

Так, x3d[i][j][k] означает то же самое, что и *(x3d[i][j]+k). Первое подвыражение, x3d[i][j], согласно А7.1, приводится к типу "указатель на массив целых"; по А7.7 сложение включает умножение на размер объекта типа int. Из этих же правил следует, что массивы запоминаются "построчно" (последние индексы меняются чаще) и что первая размерность в объявлении помогает определить количество памяти, занимаемой массивом, однако в вычислении адреса элемента массива участия не принимает.

А8.6.3. Объявители функций

В новом способе объявление функции T D, где 0 имеет вид

D1(*список-типов-параметров*)

и тип идентификатора объявления T D1 есть *"модификатор-типа* T", тип идентификатора в D есть *"модификатор-типа* функция с аргументами *список-типов-параметров,* возвращающая T". Параметры имеют следующий синтаксис:

список-типов-параметров:
  список-параметров
  список-параметров , ...
список-параметров:
  объявление -параметра
  список-параметров , *объявление-параметра*
объявление-параметра:
  спецификаторы-объявления объявитель
  спецификаторы-объявления абстрактный-объявитель$_{необ}$

При новом способе объявления функций список параметров специфици-' рует их типы, а если функция вообще не имеет параметров, на месте списка типов указывается одно слово - void. Если список типов параметров заканчивается многоточием ", ...", то функция может иметь больше аргументов, чем число явно описанных параметров. (См. А7.3.2.)

Типы параметров, являющихся массивами и функциями, заменяются на указатели в соответствии с правилами преобразования параметров

(А10.1). Единственный спецификатор класса памяти, который разрешается мспользовать в объявлении параметра, - это register, однако он игнорируется, если объявитель функции не является заголовком ее определения. Аналогично, если объявители в объявлениях параметров содержат идентификаторы, а объявитель функции не является заголовком определения функции, то эти идентификаторы тотчас же выводятся из текущей области видимости.

При старом способе объявление функции T D, где D имеет вид

D1 (*список-идентификаторов*$_{необ.}$)

и тип идентификатора объявления T D1 есть *"модификатор-типа* T", тип идентификатора в D есть *"модификатор-типа* функция от неспецифицированных аргументов, возвращающая T". Параметры, если они есть, имеют следующий вид:

список-идентификаторов:
идентификатор
список-идентификаторов , *идентификатор*

При старом способе, если объявитель функции не используется в качестве заголовка определения функции (А10.1), список идентификаторов должен отсутствовать. Никакой информации о типах параметров в объявлениях не содержится.

Например, объявление

```
int f(), *fpi(), (*pfi)();
```

объявляет функцию f, возвращающую число типа int, функцию fpi, возвращающую указатель на число типа int, и указатель pfi на функцию, возвращающую число типа int. Ни для одной функции в объявлении не указаны типы параметров; все функции описаны старым способом.

Вот как выглядит объявление в новой записи:

```
int strcpy(char *dest, const char *source), rand(void);
```

Здесь strcpy - функция с двумя аргументами, возвращающая значение типа int; первый аргумент - указатель на значение типа char, а второй - указатель на неизменяющееся значение типа char. Имена параметров играют роль хороших комментариев. Вторая функция, rand, аргументов не имеет и возвращает int.

Объявители функций с прототипами параметров - наиболее важное нововведение ANSI-стандарта. В сравнении со старым способом, принятым в первой редакции языка, они позволяют проверять и приводить к нужному типу аргументы во всех вызовах. Следует однако отметить, что их введение привнесло в язык некоторую сумятицу и необходимость согласования

обеих форм. Чтобы обеспечить совместимость, потребовались некоторые "синтаксические уродства", а именно void, для явного указания на отсутствие параметров.

Многоточие ", ..." применительно к функциям с варьируемым числом аргументов - также новинка, которая вместе со стандартным заголовочным файлом макросов <stdarg.h> формализует неофициально используемый, но официально запрещенный в первой редакции механизм.

Указанные способы записи заимствованы из языка Си++.

А 8.7. Инициализация

С помощью иниц-объявителя можно указать начальное значение объявляемого объекта. Инициализатору, представляющему собой выражение или список инициализаторов, заключенный в фигурные скобки, предшествует знак =. Этот список может завершаться запятой; ее назначение - сделать форматирование более четким.

инициализатор:
выражение-присваивания
{ *список-инициализаторов* }
{ *список-инициализаторов* , }
список-инициализаторов:
инициализатор
список-инициализаторов , *инициализатор*

В инициализаторе статического объекта или массива все выражения должны быть константными (А7.19). Если инициализатор auto- и register-объекта или массива находится в списке, заключенном в фигурные скобки, то входящие в него выражения также должны быть константными. Однако в случае автоматического объекта с одним выражением инициализатор не обязан быть константным выражением, он просто должен иметь соответствующий объекту тип.

В первой редакции не разрешалась инициализация автоматических структур, объединений и массивов. ANSI-стандарт позволяет это; однако, если инициализатор не может быть представлен одним простым выражением, инициализация может быть выполнена только с помощью константных конструкций.

Статический объект, инициализация которого явно не указана, инициализируется так, как если бы ему (или его элементам) присваивалась константа 0. Начальное значение автоматического объекта, явным образом неинициализированного, не определено.

Инициализатор указателя или объекта арифметического типа - это одно выражение (возможно, заключенное в фигурные скобки), которое присваивается объекту.

Инициализатор структуры - это либо выражение того же структурного типа, либо заключенные в фигурные скобки инициализаторы ее элементов, заданные по порядку. Безымянные битовые поля игнорируются и не инициализируются. Если инициализаторов в списке меньше, чем элементов, то оставшиеся элементы инициализируются нулем. Инициализаторов не должно быть больше числа элементов.

Инициализатор массива - это список инициализаторов его элементов, заключенный в фигурные скобки. Если размер массива не известен, то он считается равным числу инициализаторов, при этом тип его становится завершенным. Если размер массива известен, то число инициализаторов не должно превышать числа его элементов; если инициализаторов меньше, оставшиеся элементы обнуляются.

Как особый выделен случай инициализации массива символов. Последний можно инициализировать с помощью строкового литерала; символы инициализируют элементы массива в том порядке, как они заданы в строковом литерале. Точно так же, с помощью литерала из расширенного набора символов (А2.6), можно инициализировать массив типа `wchar_t`. Если размер массива не известен, то он определяется числом символов в строке, включая и завершающий NULL-символ; если размер массива известен, то число символов в строке, не считая завершающего NULL-символа, не должно превышать его размера.

Инициализатором объединения может быть либо выражение того же типа, либо заключенный в фигурные скобки инициализатор его первого элемента.

> В первой версии языка не позволялось инициализировать объединения. Правило "первого элемента" не отличается изяществом, однако не требует нового синтаксиса. Стандарт ANSI проясняет еще и семантику не инициализируемых явно объединений.

Введем для структуры и массива обобщенное имя: *агрегат*. Если агрегат содержит элементы агрегатного типа, то правила инициализации применяются рекурсивно. Фигурные скобки в некоторых случаях инициализации можно опускать. Если инициализатор элемента агрегата, который сам является агрегатом, начинается с левой фигурной скобки, то этот подагрегат инициализируется последующим списком разделенных запятыми инициализаторов; считается ошибкой, если количество инициализаторов подагрегата превышает число его элементов. Если, однако, инициализатор подагрегата не начинается с левой фигурной скобки, то чтобы его инициализировать, нужно отсчитать соответствующее число элементов из списка; при этом остальные элементы инициализируются следующими инициализаторами агрегата, для которого данный подагрегат является частью.

Например

```
int x[] = { 1, 3, 5 };
```

объявляет и инициализирует x как одномерный массив с тремя элементами, поскольку размер не был указан, а список состоит из трех инициализаторов.

```
float y[4][3] = {
   { 1, 3, 5 },
   { 2, 4, 6 },
   { 3, 5, 7 },
};
```

представляет собой инициализацию с полным набором фигурных скобок: 1, 3 и 5 инициализируют первую строку в массиве y[0], т. е. y[0][0], y[0][1] и y[0][2]. Аналогично инициализируются следующие две строки: y[1]иy[2]. Инициализаторов не хватило на весь массив, поэтому элементы строки y[3] будут нулевыми. В точности тот же результат был бы достигнут с помощью следующего объявления:

```
float y[4][3] = {
   1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для y начинается с левой фигурной скобки, но для y[0] скобки нет, поэтому из списка будут взяты три элемента. Аналогично по три элемента будут взяты для y[1], а затем и для y[2]. В

```
float y[4][3] = {
   { 1 }, { 2 }, { 3 }, { 4 }
};
```

инициализируется первый столбец матрицы y, все же другие элементы остаются нулевыми.

Наконец,

```
char msg[] = "Синтаксическая ошибка в строке %s\n";
```

представляет собой пример массива символов, элементы которого инициализируются с помощью строки; в его размере учитывается и завершающий NULL-символ.

А 8.8. Имена типов

В ряде случаев возникает потребность в применении имени типа данных (например при явном приведении к типу, в указании типов параметров внутри объявлений функций, в аргументе оператора sizeof). Эта потребность реализуется с помощью *имени типа,* определение которого син-

таксически почти совпадает с объявлением объекта того же типа. Оно отличается от объявления лишь тем, что не содержит имени объекта.

имя-типа:
список-спецификаторов-квалификаторов абстрактный-объявитель$_{необ.}$
абстрактный-объявитель:
указатель
указатель$_{необ.}$ собственно-абстрактный-объявитель
собственно-абстрактный-объявитель:
(абстрактный-объявитель)
собственно-абстрактный-объявитель$_{необ.}$ [константное-выражение$_{необ.}$]
собственно-абстрактный-объявитель$_{необ.}$ (список-типов-параметров$_{необ.}$)

Можно указать одно-единственное место в абстрактном объявителе, где мог бы оказаться идентификатор, если бы данная конструкция была полноценным объявителем. Именованный тип совпадает с типом этого "невидимого идентификатора". Например

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

соответственно обозначают типы int, "указатель на int", "массив из трех указателей на int", "указатель на массив из неизвестного количества int", "функция неизвестного количества параметров, возвращающая указатель на int", "массив неизвестного количества указателей на функции без параметров, каждая из которых возвращает int".

А 8.9. Объявление typedef

Объявления, в которых спецификатор класса памяти есть typedef, не объявляют объектов - они определяют идентификаторы, представляющие собой имена типов. Эти идентификаторы называются typedef-именами.

typedef-имя:
идентификатор

Объявление typedef приписывает тип каждому имени своего объявителя обычным способом (см. А8.6.). С этого момента typedef-имя синтаксически эквивалентно ключевому слову спецификатора типа, обозначающему связанный с ним тип. Например, после

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

допустимы следующие объявления:

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

b принадлежит типу long, bp - типу "указатель на long", z - это структура заданного вида, а zp — принадлежит типу "указатель на такую структуру".

Объявление typedef не вводит новых типов, оно только дает имена типам, которые могли бы быть специфицированы и другим способом. Например, b имеет тот же тип, что и любой другой объект типа long.

typedef-имена могут быть перекрыты другими определениями во внутренней области видимости, но при условии, что в них присутствует указание типа. Например

```
extern Blockno;
```

не переобъявляет Blockno, а вот

```
extern int Blockno;
```

переобъявляет.

А 8.10. Эквивалентность типов

Два списка спецификаторов типа эквивалентны, если они содержат одинаковый набор спецификаторов типа с учетом синонимичности названий (например, long и int long считаются одинаковыми типами). Структуры, объединения и перечисления с разными тегами считаются разными, а каждое безтеговое объединение, структура или перечисление представляет собой уникальный тип.

Два типа считаются совпадающими, если их абстрактные объявители (А8.8) после замены всех typedef-имен их типами и выбрасывания имен параметров функций составят эквивалентные списки спецификаторов типов. При сравнении учитываются размеры массивов и типы параметров функций.

А9. Инструкции

За исключением оговоренных случаев инструкции выполняются в том порядке, как они написаны. Инструкции не имеют значений и выполняются, чтобы произвести определенные действия. Все виды инструкций можно разбить на несколько групп:

инструкция:
  помеченная-инструкция
  инструкция -выражение

составная-инструкция
инструкция-выбора
циклическая -инструкция
инструкция -перехода

А 9.1. Помеченные инструкции

Инструкции может предшествовать метка.

помеченная-инструкция:
идентификатор : *инструкция*
case *константное-выражение* : *инструкция*
default : *инструкция*

Метка, состоящая из идентификатора, одновременно служит и объявлением этого идентификатора. Единственное назначение идентификатора-метки - указать место перехода для goto. Областью видимости идентификатора-метки является текущая функция. Так как метки имеют свое собственное пространство имен, они не "конфликтуют" с другими идентификаторами и не могут быть перекрыты (см. А11.1.).

case-метки и default-метки используются в инструкции switch (А9.4). Константное выражение в case должно быть целочисленным.

Сами по себе метки не изменяют порядка вычислений.

А 9.2. Инструкция-выражение

Наиболее употребительный вид инструкции – это инструкция-выражение.

инструкция-выражение:
выражение$_{необ}$;

Чаще всего инструкция-выражение - это присваивание или вызов функции. Все действия, реализующие побочный эффект выражения, завершаются, прежде чем начинает выполняться следующая инструкция. Если выражение в инструкции опущено, то она называется пустой; пустая инструкция часто используется для обозначения пустого тела циклической инструкции или в качестве места для метки.

А 9.3. Составная инструкция

Так как в местах, где по синтаксису полагается одна инструкция, иногда возникает необходимость выполнить несколько, предусматривается возможность задания составной инструкции (которую также называют блоком). Тело определения функции есть составная инструкция:

составная -инструкция:
{ *список-объявлений* *список-инструкций*$_{необ}$ }

список-объявлений:
объявление
список-объявлений объявление
список-инструкций:
инструкция
список-инструкций инструкция

Если идентификатор из списка объявлений находился в области видимости объемлющего блока, то действие внешнего объявления при входе внутрь данного блока приостанавливается (A11.1), а после выхода из него возобновляется. Внутри блока идентификатор может быть объявлен только один раз. Для каждого отдельного пространства имен эти правила действуют независимо (A11); идентификаторы из разных пространств имен всегда различны.

Инициализация автоматических объектов осуществляется при каждом входе в блок и продолжается по мере продвижения по объявителям. При передаче управления внутрь блока никакие инициализации не выполняются. Инициализации статических объектов осуществляются только один раз перед запуском программы.

А 9.4. Инструкции выбора

Инструкции выбора осуществляют отбор одной из нескольких альтернатив, определяющих порядок выполнения инструкций.

инструкция-выбора:
`if` (*выражение*) *инструкция*
if (*выражение*) *инструкция else инструкция*
switch (*выражение*) *инструкция*

Оба вида `if`-инструкций содержат выражение, которое должно иметь арифметический тип или тип указателя. Сначала вычисляется выражение со всеми его побочными эффектами, результат сравнивается с 0. В случае несовпадения с 0 выполняется первая подинструкция. В случае совпадения с 0 для второго типа if выполняется вторая подинструкция. Связанная со словом else неоднозначность разрешается тем, что слово `else` соотносят с последней еще не имеющей else `if`-инструкцией, расположенной в одном с этим `else` блоке и на одном уровне вложенности блоков.

Инструкция switch вызывает передачу управления на одну из нескольких инструкций в зависимости от значения выражения, которое должно иметьцелочисленныйтип.

Управляемая с помощью switch подинструкция обычно составная. Любая инструкция внутри этой подинструкции может быть помечена

одной или несколькими case-метками (А9.1). Управляющее выражение подвергается целочисленному повышению (А6.1), а case-константы приводятся к повышенному типу. После такого преобразования никакие две case-константы в одной инструкции switch не должны иметь одинаковых значений. Со switch-инструкцией может быть связано не более одной default-метки. Конструкции switch допускается вкладывать друг в друга; case и default-метки относятся к самой внутренней switch-инструкции из тех, которые их содержат.

Инструкция switch выполняется следующим образом. Вычисляется выражение со всеми побочными эффектами, и результат сравнивается с каждой case-константой. Если одна из case-констант равна значению выражения, управление переходит на инструкцию с соответствующей case-меткой. Если ни с одной из case-констант нет совпадения, управление передается на инструкцию с default-меткой, если такая имеется, в противном случае ни одна из подинструкций switch не выполняется.

> В первой версии языка требовалось, чтобы выражение и case-константы в switch были типа int.

А 9.5. Циклические инструкции

Циклические инструкции специфицируют циклы.

циклическая-инструкция:
 while (*выражение*) *инструкция*
 do *инструкция* while (*выражение*)
 for (*выражение*$_{необ}$; *выражение* $_{необ}$; *выражение*$_{необ}$) *инструкция*

В инструкциях while и do выполнение подинструкций повторяется до тех пор, пока значение выражения не станет нулем. Выражение должно иметь арифметический тип или тип указателя. В while вычисление выражения со всеми побочными эффектами и проверка осуществляются перед каждым выполнением инструкции, а в do — после.

В инструкции for первое выражение вычисляется один раз, тем самым осуществляется инициализация цикла. На тип этого выражения никакие ограничения не накладываются. Второе выражение должно иметь арифметический тип или тип указателя; оно вычисляется перед каждой итерацией. Как только его значение становится равным 0, for прекращает свою работу. Третье выражение вычисляется после каждой итерации и, следовательно, выполняет повторную инициализацию цикла. Никаких ограничений на его тип нет. Побочные эффекты всех трех выражений заканчиваются по завершении их вычислений. Если подинструкция не содержит в себе continue, то

 for (*выражение1* ; *выражение2* ; *выражение3*) *инструкция*

эквивалентно конструкции

```
выражение1 ;
    while ( выражение2) {
        инструкция
        выражение3 ;
}
```

Любое из трех выражений цикла может быть опущено. Считается, что отсутствие второго выражения равносильно сравнению с нулем ненулевой константы.

А 9.6. Инструкции перехода

Инструкции перехода осуществляют безусловную передачу управления.

```
инструкция-перехода:
    goto идентификатор ;
    continue ;
    break ;
    return выражение_необ;
```

В goto-инструкции идентификатор должен быть меткой (А9.1), расположенной в текущей функции. Управление передается на помеченную инструкцию.

Инструкцию continue можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. Говоря более точно, для каждой из конструкций

```
while (...) {        do {                 for (...) {
  ...                  ...                  ...
contin: ;            contin: ;            contin: ;
}                    } while (...);       }
```

инструкция continue, если она не помещена в еще более внутренний цикл, делает то же самое, что и goto contin.

Инструкция break встречается в циклической или в switch-инструкции, и только в них. Она завершает работу самой внутренней циклической или switch-инструкции, содержащей данную инструкцию break, после чего управление переходит к следующей инструкции.

С помощью return функция возвращает управление в программу, откуда была вызвана. Если за return следует выражение, то его значение возвращается вызвавшей эту функцию программе. Значение выражения приводится к типу так, как если бы оно присваивалось переменной, имеющей тот же тип, что и функция.

Ситуация, когда "путь" вычислений приводит в конец функции (т. е. на последнюю закрывающую фигурную скобку), равносильна выполнению retuгп-инструкции без выражения. При этом, а также в случае явного задания retu rn без выражения возвращаемое значение не определено.

А 10. Внешние объявления

То, что подготовлено в качестве ввода для Си-компилятора, называется единицей трансляции. Она состоит из последовательности внешних объявлений, каждое из которых представляет собой либо объявление, либо определение функции.

единица -трансляции:
внешнее-объявление
единица-трансляции внешнее-объявление
внешнее -объявление:
определение-функции
объявление

Область видимости внешних объявлений простирается до конца единицы трансляции, в которой они объявлены, точно так же, как область видимости объявлений в блоке распространяется до конца этого блока. Синтаксис внешнего объявления не отличается от синтаксиса любого другого объявления за одним исключением: код функции можно определять только с помощью внешнего объявления.

А 10.1. Определение функции

Определение функции имеет следующий вид:

определение-функции:
спецификаторы-объявления$_{необ}$ объявитель список-объявлений$_{необ}$
составная-инструкция

Из спецификаторов класса памяти в спецификаторах-объявлениях возможны только extern и static; различия между последними рассматриваются в А11.2.

Типом возвращаемого функцией значения может быть арифметический тип, структура, объединение, указатель и void, но не "функция" и не "массив". Объявитель в объявлении функции должен явно указывать на то, что описываемый им идентификатор имеет тип "функция", т. е. он должен иметь одну из следующих двух форм (А8.6.3):

собственно-объявитель (список-типов-параметров)
собственно-объявитель (список-идентификаторов$_{необ}$)

10 Зак. 1116

где собственно-объявитель есть идентификатор или идентификатор, заключенный в скобки. Заметим, что тип "функция" посредством typedef получить нельзя.

Первая форма соответствует определению функции новым способом, для которого характерно объявление параметров в списке-типов-параметров вместе с их типами; за объявителем не должно быть списка-объявлений. Если список-типов-параметров не состоит из одного-единственного слова void, показывающего, что параметров у функции нет, то в каждом объявителе в списке-типов-параметров обязан присутствовать идентификатор. Если список-типов-параметров заканчивается знаками ", ... ", то вызов функции может иметь аргументов больше, чем параметров; в таком случае, чтобы обращаться к дополнительным аргументам, следует пользоваться механизмом макроса va_arg из заголовочного файла <stdarg.h>, описанного в приложении В. Функции с переменным числом аргументов должны иметь по крайней мере один именованный параметр.

Вторая форма - определение функции старым способом. Список-идентификаторов содержит имена параметров, а список-объявлений приписывает им типы. В списке-объявлении разрешено объявлять только именованные параметры, инициализация запрещается, и из спецификаторов класса памяти возможен только register.

И в том и другом способе определения функции мыслится, что все параметры как бы объявлены в самом начале составной инструкции, образующей тело функции, и совпадающие с ними имена здесь объявляться не должны (хотя, как и любые идентификаторы, их можно переобъявить в более внутренних блоках). Объявление параметра "массив из *типа*" можно трактовать как "указатель на *тип*"; аналогично объявлению параметра объявление "функция, возвращающая *тип*" можно трактовать как "указатель на функцию, возвращающую *тип*". В момент вызова функции ее аргументы соответствующим образом преобразуются и присваиваются параметрам (см. А7.3.2).

> Новый способ определения функций введен ANSI-стандартом. Есть также небольшие изменения в операции повышения типа; в первой версии языка параметры типа float следовало читать как double. Различие между float и double становилось заметным, лишь когда внутри функции генерировался указатель на параметр.

Ниже приведен пример определения функции новым способом:

```
int max(int a, int b, int c)
{
  int m;

  m = (a > b) ? a : b;
```

```
    return (m > c) ? m : c;
}
```

Здесь int — спецификаторы-объявления; max(int a, int b, int c) – объявитель функции, а { ... } - блок, задающий ее код. Определение старым способом той же функции выглядит следующим образом:

```
int max(a, b, c)
int a, b, c;
{
        /* ... */
}
```

где max(a, b, c) – объявитель, а int a, b, c - список-объявлений для параметров.

А 10.2. Внешние объявления

Внешние объявления специфицируют характеристики объектов, функций и других идентификаторов. Термин "внешний" здесь используется, чтобы подчеркнуть тот факт, что объявления расположены вне функций; впрямую с ключевым словом extern ("внешний") он не связан. Класс памяти для объекта с внешним объявлением либо вообще не указывается, либо специфицируется как extern или static.

В одной единице трансляции для одного идентификатора может содержаться несколько внешних объявлений, если они согласуются друг с другом по типу и способу связи и если для этого идентификатора существует не более одного определения.

Два объявления объекта или функции считаются согласованными по типу в соответствии с правилами, рассмотренными в А8.10. Кроме того, если объявления отличаются лишь тем, что в одном из них тип структуры, объединения или перечисления незавершен (А8.3), а в другом соответствующий ему тип с тем же тегом завершен, то такие типы считаются согласованными. Если два типа массива (8.6.2) отличаются лишь тем, что один завершенный, а другой незавершенный, то такие типы также считаются согласованными. Наконец, если один тип специфицирует функцию старым способом, а другой - ту же функцию новым способом (с объявлениями параметров), то такие типы также считаются согласованными.

Если первое внешнее объявление функции или объекта помечено спецификатором static, то объявленный идентификатор имеет *внутреннюю связь*; в противном случае - *внешнюю связь*. Способы связей обсуждаются в А11.2.

Внешнее объявление объекта считается определением, если оно имеет инициализатор. Внешнее объявление, в котором нет инициализатора и нет

спецификатора extern, считается *пробным определением.* Если в единице трансляции появится определение объекта, то все его пробные определения просто станут избыточными объявлениями. Если никакого определения для этого объекта в единице трансляции не обнаружится, то все его пробные определения будут трактоваться как одно определение с инициализатором 0.

Каждый объект должен иметь ровно одно определение. Для объекта с внутренней связью это правило относится к каждой отдельной единице трансляции, поскольку объекты с внутренними связями в каждой единице уникальны. В случае объектов с внешними связями указанное правило действует в отношении всей программы в целом.

> Хотя правило одного определения формулируется несколько иначе, чем в первой версии языка, по существу оно совпадает с прежним. Некоторые реализации его ослабляют, более широко трактуя понятие пробного определения. В другом варианте указанного правила, который распространен в системах UNIX и признан как общепринятое расширение стандарта, все пробные определения объектов с внешними связями из всех транслируемых единиц программы рассматриваются вместе, а не отдельно в каждой единице. Если где-то в программе обнаруживается определение, то пробные определения становятся просто объявлениями, но, если никакого определения не встретилось, то все пробные определения становятся одним-единственным определением с инициализатором 0.

А 11. Область видимости и связи

Каждый раз компилировать всю программу целиком нет необходимости. Исходный текст можно хранить в нескольких файлах, представляющих собой единицы трансляции. Ранее скомпилированные программы могут загружаться из библиотек. Связи между функциями программы могут осуществляться через вызовы и внешние данные.

Следовательно, существуют два вида областей видимости: первая - это *лексическая область* идентификатора: т. е. область в тексте программы, где имеют смысл все его характеристики; вторая область - это область, ассоциируемая с объектами и функциями, имеющими внешние связи, устанавливаемые между идентификаторами из раздельно компилируемых единиц трансляции.

А 11.1. Лексическая область видимости

Каждый идентификатор попадает в одно из нескольких пространств имен. Эти пространства никак не связаны друг с другом. Один и тот же идентификатор может использоваться в разных смыслах даже в одной об-

ласти видимости, если он принадлежит разным пространствам имен. Ниже через точку с запятой перечислены классы объектов, имена которых представляют собой отдельные независимые пространства: объекты, функции, typedef-имена и enum-константы; метки инструкций; теги структур, объединений и перечислений; элементы каждой отдельной структуры или объединения.

> Сформулированные правила несколько отличаются от прежних, описанных в первом издании. Метки инструкций не имели раньше собственного пространства; теги структур и теги объединений (а в некоторых реализациях и теги перечислений) имели отдельные пространства. Размещение тегов структур, объединений и перечислений в одном общем пространстве - это дополнительное ограничение, которого раньше не было. Наиболее существенное отклонение от первой редакции в том, что каждая отдельная структура (или объединение) создает свое собственное пространство имен для своих элементов. Таким образом, одно и то же имя может использоваться в нескольких различных структурах. Это правило широко применяется уже несколько лет.

Лексическая область видимости идентификатора объекта (или функции), объявленного во внешнем объявлении, начинается с места, где заканчивается его объявитель, и простирается до конца единицы трансляции, в которой он объявлен. Область видимости параметра в определении функции начинается с начала блока, представляющего собой тело функции, и распространяется на всю функцию; область видимости параметра в описании функции заканчивается в конце этого описания. Область видимости идентификатора, объявленного в начале блока, начинается от места, где заканчивается его объявитель, и продолжается до конца этого блока. Областью видимости метки является вся функция, где эта метка встречается. Область видимости тега структуры, объединения или перечисления начинается от его появления в спецификаторе типа и продолжается до конца единицы трансляции для объявления внешнего уровня и до конца блока для объявления внутри функции.

Если идентификатор явно объявлен в начале блока (в том числе тела функции), то любое объявление того же идентификатора, находящееся снаружи этого блока, временно перестает действовать вплоть до конца блока.

А 11.2. Связи

Если встречается несколько объявлений, имеющих одинаковый идентификатор и описывающих объект (или функцию), то все эти объявления в случае внешней связи относятся к одному объекту (функции) - уникальному для всей программы; если же связь внутренняя, то свойство уникальности распространяется только на единицу трансляции.

Как говорилось в А10.2, если первое внешнее объявление имеет спецификатор static, то оно описывает идентификатор с внутренней связью, если такого спецификатора нет, то - с внешней связью. Если объявление находится внутри блока и не содержит extern, то соответствующий идентификатор ни с чем не связан и уникален для данной функции. Если объявление содержит extern и блок находится в области видимости внешнего объявления этого идентификатора, то последний имеет ту же связь и относится к тому же объекту (функции). Однако если ни одного внешнего объявления для этого идентификатора нет, то он имеет внешнюю связь.

А 12. Препроцессирование

Препроцессор выполняет макроподстановку, условную компиляцию, включение именованных файлов. Строки, начинающиеся со знака # (перед которым возможны символы-разделители), устанавливают связь с препроцессором. Их синтаксис не зависит от остальной части языка; они могут появляться где угодно и оказывать влияние (независимо от области видимости) вплоть до конца транслируемой единицы. Границы строк принимаются во внимание; каждая строка анализируется отдельно (однако есть возможность "склеивать" строки, см. А12.2). Лексемами для препроцессора являются все лексемы языка и последовательности символов, задающие имена файлов, как, например, в директиве #include (А12.4). Кроме того, любой символ, не определенный каким-либо другим способом, воспринимается как лексема. Влияние символов-разделителей, отличающихся от пробелов и горизонтальных табуляций, внутри строк препроцессора не определено.

Само препроцессирование проистекает в нескольких логически последовательных фазах. В отдельных реализациях некоторые фазы объединены.

1. Трехзнаковые последовательности, описанные в А12.1, заменяются их эквивалентами. Между строками вставляются символы новой строки, если того требует операционная система.
2. Выбрасываются пары символов, состоящие из обратной наклонной черты с последующим символом новой строки; тем самым осуществляется "склеивание" строк (А12.2).
3. Программа разбивается на лексемы, разделенные символами-разделителями. Комментарии заменяются единичными пробелами. Затем выполняются директивы препроцессора и макроподстановки (А12.3–А12.10).

4. Эскейп-последовательности в символьных константах и строковых литералах (А2.5.2, А2.6) заменяются на символы, которые они обозначают. Соседние строковые литералы конкатенируются.
5. Результат транслируется. Затем устанавливаются связи с другими программами и библиотеками посредством сбора необходимых программ и данных и соединения ссылок на внешние функции и объекты с их определениями.

А 12.1. Трехзнаковые последовательности

Множество символов, из которых набираются исходные Си-программы, основано на семибитовом ASCII-коде. Однако он шире, чем инвариантный код символов ISO 646–1983 (ISO 646–1983 Invariant Code Set). Чтобы дать возможность пользоваться сокращенным набором символов, все указанные ниже трехзнаковые последовательности заменяются на соответствующие им единичные символы. Замена осуществляется до любой иной обработки.

??=	#	??(	[	??<	{
??/	\	??)	[	??>	}
??'	^	??!	,'	??-	~

Никакие другие замены, кроме указанных, не делаются.

Трехзнаковые последовательности введены ANSI-стандартом.

А 12.2. Склеивание строк

Строка, заканчивающаяся обратной наклонной чертой, соединяется со следующей, поскольку символ \ и следующий за ним символ новой строки выбрасываются. Это делается перед "разбиением" текста на лексемы.

А 12.3. Макроопределение и макрорасширение

Управляющая строка вида

`# define` *идентификатор последовательность-лексем*

заставляет препроцессор заменять идентификатор на последовательность лексем; символы-разделители в начале и в конце последовательности лексем выбрасываются. Повторная строка tfdefine с тем же идентификатором считается ошибкой, если последовательности лексем неидентичны (несовпадения в символах-разделителях при сравнении во внимание не принимаются). Строка вида

`# define` *идентификатор*(*список-идентификаторов*)
последовательность-лексем

где между первым идентификатором и знаком (не должно быть ни одного

символа-разделителя, представляет собой макроопределение с параметрами, задаваемыми списком идентификаторов. Как и в первом варианте, символы-разделители в начале и в конце последовательности лексем выбрасываются, и макрос может быть повторно определен только с тем же списком параметров и с той же последовательностью лексем. Управляющая строка вида

undef *идентификатор*

предписывает препроцессору "забыть" определение, данное идентификатору. Применение #undef к неизвестному идентификатору ошибкой не считается.

Если макроопределение было задано вторым способом, то текстовая последовательность, состоящая из его идентификатора, возможно, со следующими за ним символами-разделителями, знака (, списка лексем, разделенных запятыми, и знака), представляет собой вызов макроса. Аргументами вызова макроса являются лексемы, разделенные запятыми (запятые, "закрытые" кавычками или вложенными скобками, в разделении аргументов не участвуют). Аргументы при их выделении макрорасширениям не подвергаются. Количество аргументов в вызове макроса должно соответствовать количеству параметров макроопределения. После выделения аргументов окружающие их символы-разделители выбрасываются. Затем в замещающей последовательности лексем макроса идентификаторы-параметры (если они не закавычены) заменяются на соответствующие им аргументы. Если в замещающей последовательности перед параметром не стоит знак # и ни перед ним, ни после него нет знака ##, то лексемы аргумента проверяются: не содержат ли они в себе макровызова, и если содержат, то прежде чем аргумент будет подставлен, производится соответствующее ему макрорасширение.

На процесс подстановки влияют два специальных оператора. Первый - это оператор #, который ставится перед параметром. Он требует, чтобы подставляемый вместо параметра и знака # (перед ним) текст был заключен в двойные кавычки. При этом в строковых литералах и символьных константах аргумента перед каждой двойной кавычкой " (включая и обрамляющие строки), а также перед каждой обратной наклонной чертой \ вставляется \.

Второй оператор записывается как ##. Если последовательность лексем в любого вида макроопределении содержит оператор *tttt,* то сразу после подстановки параметров он вместе с окружающими его символами-разделителями выбрасывается, благодаря чему "склеиваются" соседние лексемы, образуя тем самым новую лексему. Результат не определен при получении неправильных лексем или когда генерируемый текст зависит

от порядка применения операторов ##. Кроме того, ## не может стоять ни в начале, ни в конце замещающей последовательности лексем.

В макросах обоих видов замещающая последовательность лексем повторно просматривается на предмет обнаружения там новых define-имен. Однако, если некоторый идентификатор уже был заменен в данном расширении, повторное появление такого идентификатора не вызовет его замены.

Если полученное расширение начинается со знака #, оно не будет воспринято как директива препроцессора.

> В ANSI-стандарте процесс макрорасширения описан более точно, чем в первом издании книги. Наиболее важные изменения касаются введения операторов tt и tt#, которые предоставляют возможность осуществлять расширения внутри строк и конкатенацию лексем. Некоторые из новых правил, особенно касающиеся конкатенации, могут показаться несколько странными. (См. приведенные ниже примеры.)

Описанные возможности можно использовать для показа смысловой сущности констант, как, например, в

```
#define TABSIZE 100
int  table[TABSIZE];
```

Определение

```
«define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

задает макрос, возвращающий абсолютное значение разности его аргументов. В отличие от функции, делающей то же самое, аргументы и возвращаемое значение здесь могут иметь любой арифметический тип и даже быть указателями. Кроме того, аргументы, каждый из которых может иметь побочный эффект, вычисляются дважды: один раз - при проверке, другой раз - при вычислении результата.

Если имеется определение

```
#define tempfile(dir) #dir "/%s"
```

то макровызов tempfile(/usr/tmp) даст в результате

```
"/usr/tmp" "/%s"
```

Далее эти две строки превратятся в одну строку. По макросу

```
#define cat(x,y)  x «« y
```

вызов cat(var, 123) сгенерирует var123. Однако cat (cat (1,2), 3) не даст желаемого, так как оператор ## воспрепятствует получению правильных

аргументов для внешнего вызова cat. В результате будет выдана следующая цепочка лексем:

```
cat ( 1 , 2 )3
```

где)3 (результат "склеивания" последней лексемы первого аргумента с первой лексемой второго аргумента) не является правильной лексемой. Если второй уровень макроопределения задан в виде

```
#define xcat(x,y) cat(x,y)
```

то никаких коллизий здесь не возникает; xcat(xcat(1, 2), 3) в итоге даст 123, поскольку сам xcat не использует оператора *tttt*.

Аналогично сработает и ABSDIFF(ABSDIFF(a,b),c), и мы получим правильный результат.

А 12.4. Включение файла

Управляющая строка

include <*имя-файла*>

заменяется на содержимое файла с именем *имя-файла.* Среди символов, составляющих *имя-файла,* не должно быть знака > и символа новой строки. Результат не определен, если *имя-файла* содержит любой из символов ", ', \ или пару символов /*. Порядок поиска указанного файла зависит от реализации.

Подобным же образом выполняется управляющая строка

« include *"имя-файла"*

Сначала поиск осуществляется по тем же правилам, по каким компилятор ищет первоначальный исходный файл (механизм этого поиска зависит от реализации), а в случае неудачи осуществляется методом поиска, принятым в #include первого типа. Результат остается неопределенным, если имя файла содержит ", \ или /*; использование знака > разрешается.

Наконец, директива

« include *последовательность-лексем*

не совпадающая ни с одной из предыдущих форм, рассматривает последовательность лексем как текст, который в результате всех макроподстановок должен дать #include <...> или #include "...". Сгенерированная таким образом директива далее будет интерпретироваться в соответствии с полученной формой.

Файлы, вставляемые с помощью #include, сами могут содержать в себе директивы #include.

А 12.5. Условная компиляция

Части программы могут компилироваться условно, если они оформлены в соответствии со следующим схематично изображенным синтаксисом:

условная-конструкция-препроцессора:
if-строка текст elif-части else-часть$_{необ.}$ #endif
if-строка:
if *константное-выражение*
ifdef *идентификатор*
tfifndef *идентификатор*
elif-части:
elif-строка текст
elif-части $_{необ.}$
elif-строка:
elif *константное-выражение*
else-часть:
else-строка текст
else-строка:
else

Каждая из директив (if-строка, elif-строка, else-строка и ttendif) записывается на отдельной строке. Константные выражения в #if и последующих строках flelif вычисляются по порядку, пока не обнаружится выражение с ненулевым (истинным) значением; текст, следующий за строкой с нулевым значением, выбрасывается. Текст, расположенный за директивой с ненулевым значением, обрабатывается обычным образом. Под словом "текст" здесь имеется в виду любая последовательность строк, включая строки препроцессора, которые не являются частью условной структуры; текст может быть и пустым. Если строка #if или #elif с ненулевым значением выражения найдена и ее текст обработан, то последующие строки tfelif и flelse вместе со своими текстами выбрасываются. Если все выражения имеют нулевые значения и присутствует строка flelse, то следующий за ней текст обрабатывается обычным образом. Тексты "неактивных" ветвей условных конструкций, за исключением тех, которые заведуют вложенностью условных конструкций, игнорируются.

Константные выражения в flif и #elif являются объектами для обычной макроподстановки. Более того, прежде чем просматривать выражения вида

defined *идентификатор*

и

defined (*идентификатор*)

на предмет наличия в них макровызова, они заменяются на 1L или OL в зависимости от того, был или не был определен препроцессором указанный в них идентификатор. Все идентификаторы, оставшиеся после макрорасширения, заменяются на OL. Наконец, предполагается, что любая целая константа всегда имеет суффикс L, т. е. вся арифметика имеет дело с операндами только типа long или unsigned long.

Константное выражение (A7.19) здесь используется с ограничениями: оно должно быть целочисленным, не может содержать в себе перечислимых констант, преобразований типа и операторов sizeof.

Управляющие строки

#ifdef *идентификатор*
#ifndef *идентификатор*

эквивалентны соответственно строкам

if defined *идентификатор*
if ! defined *идентификатор*

Строки #elif не было в первой версии языка, хотя она и использовалась в некоторых препроцессорах. Оператор препроцессора defined - также новый.

A 12.6. Нумерация строк

Для удобства работы с другими препроцессорами, генерирующими Си-программы, можно использовать одну из следующих директив:

line *константа "имя файла"*
line *константа*

Эти директивы предписывают компилятору считать, что указанные десятичное целое и идентификатор являются номером следующей строки и именем текущего файла соответственно. Если имя файла отсутствует, то ранее запомненное имя не изменяется. Расширения макровызовов в директиве tfline выполняются до интерпретации последней.

A 12.7. Генерация сообщения об ошибке

Строка препроцессора вида

error *последовательность-лексем*$_{необ.}$

приказывает ему выдать диагностическое сообщение, включающее заданную последовательность лексем.

A 12.8. Прагма

Управляющая строка вида

pragma *последовательность-лексем*$_{необ}$

призывает препроцессор выполнить зависящие от реализации действия. Неопознанная прагма игнорируется.

А 12.9. Пустая директива

Строка препроцессора вида

```
#
```

не вызывает никаких действий.

А 12.10. Заранее определенные имена

Препроцессор "понимает" несколько заранее определенных идентификаторов; их он заменяет специальной информацией. Эти идентификаторы (и оператор препроцессора defined в том числе) нельзя повторно переопределять, к ним нельзя также применять директиву #undef. Это следующие идентификаторы:

__LINE__ Номер текущей строки исходного текста, десятичная константа.
__FILE__ Имя компилируемого файла, строка.
__DATE__ Дата компиляции в виде "Ммм дд гггг", строка.
__TIME__ Время компиляции в виде "чч:мм:сс", строка.
__STDC__ Константа 1. Предполагается, что этот идентификатор определен как 1 только в тех реализациях, которые следуют стандарту.

Строки #error и #pragma впервые введены ANSI-стандартом. Заранее определенные макросы препроцессора также до сих пор не описывались, хотя и использовались в некоторых реализациях.

А 13. Грамматика

Ниже приведены грамматические правила, которые мы уже рассматривали в данном приложении. Они имеют то же содержание, но даны в ином порядке.

Здесь не приводятся определения следующих символов-терминов: *целая-константа, символьная-константа, константа-с-плавающей-точкой, идентификатор, строка* и *константа-перечисление*. Слова, набранные обычным латинским шрифтом (не курсивом), и знаки рассматриваются как символы-термины и используются точно в том виде, как записаны. Данную грамматику можно механически трансформировать в текст, понятный системе автоматической генерации грамматического распознавателя. Для этого помимо добавления некоторых синтаксических пометок, предназначенных для указания альтернативных продукций, потребуется расшифровка

конструкции со словами "один из" и дублирование каждой продукции, использующей символ с индексом *необ.,* причем один вариант продукции должен быть написан *с* этим символом, а другой - без него. С одним изменением, а именно - удалением продукции *typedef-имя: идентификатор* и объявлением *typedef-имени* символом-термином, данная грамматика будет понятна генератору грамматического распознавателя YACC. Ей присуще лишь одно противоречие, вызываемое неоднозначностью конструкции `if-else`.

единица-трансляции:
внешнее- объявление
единица-трансляции внешнее-объявление

внешнее-объявление:
определение-функции
объявление

определение функции:
спецификаторы -объявления$_{необ}$ объявитель
список-объявлений $_{необ}$ составная-инструкция

объявление:
спецификаторы-объявления список-инициализаторов-объявителей$_{необ}$

список-объявлений:
объявление
список-объявлений объявление

спецификаторы-объявления:
спецификатор-класса-памяти спецификаторы-объявления$_{необ}$
спецификатор-типа спецификаторы-объявления$_{необ}$
квалификатор-типа спецификаторы-объявления $_{необ}$

спецификатор-класса-памяти: один из
auto register static extern typedef

спецификатор-типа: один из
void char short int long float double signed unsigned
спецификатор-структуры-или-объединения спецификатор-перечисления typedef-имя

квалификатор-типа: один из
const volatile

спецификатор-структуры -или-объединения:
структуры-или-объединения идентификатор$_{необ}$ { список-объявлений-структуры}
структуры-или-объединения идентификатор

структура-или-объединение: одно из
struct union

список-объявлений-структуры:
объявление-структуры
список-объявлений-структуры объявление-структуры

список-объявителей-инициализаторов:
объявитель-инициализатор
список-объявителей-инициализаторов , *объявитель-инициализатор*

объявитель-инициализатор:
объявитель
объявитель = *инициализатор*

объявление-структуры:
список-спецификаторов-квалификаторов список-объявителей-структуры

список-спецификаторов-квалификаторов:
спецификатор-типа список-спецификаторов-квалификаторов$_{необ.}$
квалификатор-типа список-спецификаторов-квалификаторов$_{необ.}$

список-структуры-объявителей:
структуры-объявитель
список-структуры-объявителей , *структуры-объявитель*

структуры-объявитель:
объявитель
объявитель$_{необ}$: *константное-выражение*

спецификатор-перечисления:
enum *идентификатор*$_{необ}$ { *список-перечислителей* }
enum *идентификатор*

список-перечислителей:
перечислитель
список-перечислителей перечислитель

перечислитель:
идентификатор
указатель$_{необ}$ *собственно-объявитель*

собственно -объявитель:
идентификатор
(*объявитель*)

собственно-объявитель [*константное-выражение*$_{необ}$]
собственно-объявитель (*список-типов-параметров*)
собственно-объявитель (*список-идентификаторов*$_{необ}$)

указатель:
* *список-квалификаторов-типа*$_{необ.}$
* *список-квалификаторов-типа*$_{необ.}$ *указатель*

список-квалификаторов-типа:
квалификатор-типа
список-квалификаторов-типа квалификатор-типа

список-типов-параметров:
список-параметров
список-параметров , ...

список-параметров:
объявление -параметра
список-параметров , *объявление-параметра*

объявление-параметра:
спецификаторы -объявления объявитель
спецификаторы-объявления абстрактный-объявитель$_{необ}$

список-идентификаторов:
идентификатор
список-идентификаторов , *идентификатор*

инициализатор:
выражение-присваивания
{ *список-инициализаторов* }
{ *список-инициализаторов* , }

список-инициализаторов:
инициализатор
список-инициализаторов , *инициализатор*

имя-типа:
список-спецификаторов-квалификаторов абстрактный-объявитель$_{необ.}$

абстрактный-объявитель:
указатель
указатель$_{необ.}$ *собственно-абстрактный-объявитель*

собственно-абстрактный-объявитель:
(*абстрактный-объявитель*)
собственно-абстрактный-объявитель$_{необ}$ [*константное-выражение*$_{необ.}$]
собственно-абстрактный-объявитель$_{необ}$ (*список-типов-параметров*$_{необ.}$)

typedef-имя:
идентификатор

инструкция:
помеченная-инструкция
инструкция -выражение
составная-инструкция
инструкция-выбора
циклическая -инструкция
инструкция-перехода

помеченная-инструщия:
идентификатор : *инструкция*
case *константное-выражение* : *инструкция*
default : *инструкция*

инструкция-выражение:
выражение$_{необ}$;

составная-инструщия:
(*список-объявлений*$_{необ}$ *список-инструкций*$_{необ}$)

список-инструкций:
инструкция
список-инструкций инструкция

инструкция-выбора:
if (*выражение*) *инструкция*
if (*выражение*) *инструкция* else *инструкция*
switch (*выражение*) *инструкция*

циклическая -инструкция:
while (*выражение*) *инструкция*
do *инструкция* while (*выражение*)
return *выражение*$_{необ}$;

выражение:
выражение-присваивания
выражение , *выражение-присваивания*

выражение -присваивания:
условное -выражение
унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из

= *= /= %= += -= <<= >>= &= ^= |=

условное-выражение:
логическое -ИЛИ- выражение
логическое-ИЛИ-выражение ? *выражение* : *условное-выражение*

константное-выражение:
условное-выражение

логическое -ИЛИ-выражение:
логическое-И-выражение
логическое-ИЛИ-выражение || *логическое-И-выражение*

логическое-И-выражение:
ИЛИ-выражение
логическое-И-выражение && *ИЛИ-выражение*

ИЛИ-выражение:
исключающее -ИЛИ-выражение
ИЛИ-выражение | *исключающее-ИЛИ-выражение*

исключающее-ИЛИ-выражение:
И-выражение
исключающее-ИЛИ-выражение ^ *И-выражение*

И-выражение:
выражение-равенства
И-выражение & *выражение-равенства*

выражение-равенства:
выражение-отношения
выражение-равенства == *выражение-отношения*
выражение-равенства != *выражение-отношения*

выражение-отношения:
сдвиговое -выражение
выражение-отношения < *сдвиговое-выражение*
выражение-отношения > *сдвиговое-выражение*
выражение-отношения <= *сдвиговое-выражение*
выражение-отношения >= *сдвиговое-выражение*

сдвиговое-выражение:
аддитивное-выражение
сдвиговое-выражение >> *аддитивное-выражение*
сдвиговое-выражение << *аддитивное-выражение*

аддитивное-выражение:
мультипликативное-выражение
аддитивное-выражение + *мультипликативное-выражение*
аддитивное-выражение - *мультипликативное-выражение*

мультипликативное-выражение:
выражение-приведенное-к-типу
мультипликативное-выражение * *выражение-приведенное-к-типу*
мультипликативное-выражение / *выражение-приведенное-к-типу*
мультипликативное-выражение % *выражение-приведенное-к-типу*

выражение-приведенное-к-типу:
унарное-выражение
(*имя-типа*) *выражение-приведенное-к-типу*

унарное-выражение:
постфиксное-выражение
++ *унарное-выражение*
-- *унарное-выражение*
унарный-оператор *выражение-приведенное-к-типу*
sizeof *унарное-выражение*
sizeof (*имя-типа*)

унарный-оператор: один из
& * + - ~ !

постфиксное-выражение:
первичное-выражение
постфиксное-выражение [*выражение*]
постфиксное-выражение (*список-аргументов-выражений*$_{необ.}$)
постфиксное-выражение . *идентификатор*
постфиксное-выражение -> *идентификатор*
постфиксное-выражение ++
постфиксное-выражение --

первичное-выражение:
идентификатор
константа
строка
(*выражение*)

список-аргументов-выражений:
выражение-присваивания
список-аргументов-выражений , *выражение-присваивания*

константа:
целая-константа
символьная-константа
константа-с-плавающей-точкой
константа-перечисление

Ниже приводится грамматика языка препроцессора в виде перечня структур управляющих строк. Для механического получения программы грамматического разбора она не годится. Грамматика включает символ *текст,* который означает текст обычной программы, безусловные управляющие строки препроцессора и его законченные условные конструкции.

управляющая-строка:
  # define *идентификатор последовательность-лексем*
  it define *идентификатор (идентификатор. . . . .идентификатор) последовательность-лексем*
  # undef *идентификатор*
  ft include <*имя-файла*>
  # include *"имя-файла"*
  # include *последовательность-лексем*
  # line *константа "идентификатор"*
  # line *константа*
  # error *последовательность -лексем* $_{\text{необ.}}$
  # pragma *последовательность-лексем* $_{\text{необ}}$
  #
  условная-конструкция-препроцессора

условная-конструкция-препроцессора:
  if-строкатекст elif-частиelse-часть $_{\text{необ.}}$ # endif

if-строка:
  # if *константное-выражение*
  # ifdef *идентификатор*
  # ifndef *идентификатор*

elif-части:
  elif-строка текст
  elif-части $_{\text{необ.}}$

elif-строка:
  # elif *константное-выражение*

else-часть:
  else-строка текст

else-строка:
  # else

Приложение В

Стандартная библиотека

Настоящее приложение представляет собой краткое изложение библиотеки, утвержденной в качестве ANSI-стандарта. Сама по себе библиотека не является частью языка, однако заложенный в ней набор функций, а также определений типов и макросов составляет системную среду, поддерживающую стандарт Си. Мы не приводим здесь несколько функций с ограниченной областью применения - те, которые легко синтезируются из других функций, а также опускаем все то, что касается многобайтовых символов и специфики, обусловленной языком, национальными особенностями и культурой.

Функции, типы и макросы объявляются в следующих стандартных *заголовочных файлах:*

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

Доступ к заголовочному файлу осуществляется с помощью строки препроцессора

`#include` <*заголовочный файл*>

Заголовочные файлы можно включать в любом порядке и сколько угодно раз. Строка `#include` не должна быть внутри внешнего объявления или определения и должна встретиться раньше, чем что-нибудь из включаемого заголовочного файла будет востребовано. В конкретной реализации заголовочный файл может и не быть исходным файлом.

Внешние идентификаторы, начинающиеся со знака подчеркивания, а также все другие идентификаторы, начинающиеся с двух знаков подчеркивания или с подчеркивания и заглавной буквы, зарезервированы для использования в библиотеке.

В 1. Ввод-вывод: <stdio.h>

Определенные в <stdio.h> функции ввода-вывода, а также типы и макросы составляют приблизительно одну треть библиотеки.

Поток - это источник или получатель данных; его можно связать с диском или с каким-то другим внешним устройством. Библиотека поддерживает два вида потоков: текстовый и бинарный, хотя на некоторых системах, в частности в UNIXе, они не различаются. Текстовый поток - это последовательность строк; каждая строка имеет нуль или более символов и заканчивается символом '\n'. Операционная среда может потребовать коррекции текстового потока (например перевода '\n' в символы возврат-каретки и перевод-строки).

Бинарный поток - это последовательность непреобразованных байтов, представляющих собой некоторые промежуточные данные, которые обладают тем свойством, что если их записать, а затем прочесть той же системой ввода-вывода, то мы получим информацию, совпадающую с исходной.

Поток соединяется с файлом или устройством посредством его *открытия,* указанная связь разрывается путем *закрытия* потока. Открытие файла возвращает указатель на объект типа FILE, который содержит всю информацию, необходимую для управления этим потоком. Если не возникает двусмысленности, мы будем пользоваться терминами "файловый указатель" и "поток" как равнозначными.

Когда программа начинает работу, уже открыты три потока: stdin, stdout и stderr.

В 1.1. Операции над файлами

Ниже перечислены функции, оперирующие с файлами. Тип size_t - беззнаковый целочисленный тип, используемый для описания результата оператора sizeof.

FILE *fopen(const char *filename, const char *mode)

fopen открывает файл с заданным именем и возвращает поток или NULL, если попытка открытия оказалась неудачной. Режим mode допускает следующие значения:

"r" - текстовый файл открывается для чтения (от *read* (англ.) - читать);

"w" - текстовый файл создается для записи; старое содержимое (если оно было) выбрасывается (от *write* (англ.) - писать);

"a" - текстовый файл открывается или создается для записи в конец файла (от *append* (англ.) - добавлять);

"r+" - текстовый файл открывается для исправления (т. е. для чтения и для записи);

"w+" - текстовый файл создается для исправления; старое содержимое (если оно было) выбрасывается;

"a+" - текстовый файл открывается или создается для исправления уже существующей информации и добавления новой в конец файла.

Режим "исправления" позволяет читать и писать в один и тот же файл; при переходах от операций чтения к операциям записи и обратно должны осуществляться обращения к fflush или к функции позиционирования файла. Если указатель режима дополнить буквой b (например "rb" или "w+b"), то это будет означать, что файл бинарный. Ограничение на длину имени файла задано константой FILENAME_MAX. Константа FOPEN_MAX ограничивает число одновременно открытых файлов.

FILE *freopen(const char *filename, const char *mode, FILE *stream)

freopen открывает файл с указанным режимом и связывает его с потоком stream. Она возвращает stream или, в случае ошибки, NULL. Обычно freopen используется для замены файлов, связанных с stdin, stdout или stderr, другими файлами.

int fflush(FILE *stream)

Применяемая к потоку вывода функция fflush производит дозапись всех оставшихся в буфере (еще не записанных) данных; для потока ввода эта функция не определена. Возвращает EOF в случае возникшей при записи ошибки или нуль в противном случае. Обращение вида fflush(NULL) выполняет указанные операции для всех потоков вывода.

int fclose(FILE *stream)

fclose производит дозапись еще не записанных буферизованных данных, сбрасывает несчитанный буферизованный ввод, освобождает все автоматически запрошенные буфера, после чего закрывает поток. Возвращает EOF в случае ошибки и нуль в противном случае.

int remove(const char *filename)

remove удаляет файл с указанным именем; последующая попытка открыть файл с этим именем вызовет ошибку. Возвращает ненулевое значение в случае неудачной попытки.

int rename(const char *oldname, const char *newname)

rename заменяет имя файла; возвращает ненулевое значение в случае, если попытка изменить имя оказалась неудачной. Первый параметр задает старое имя, второй - новое.

FILE *tmpfile(void)

tmpfile создает временный файл с режимом доступа "wb+", который автоматически удаляется при его закрытии или обычном завершении программой своей работы. Эта функция возвращает поток или, если не смогла создать файл, NULL.

char *tmpnam(char s[L_tmpnam])

tmpnam(NULL) создает строку, не совпадающую ни с одним из имен существующих файлов, и возвращает указатель на внутренний статический массив.

tmpnam(s) запоминает строку в s и возвращает ее в качестве значения функции; длина s должна быть не менее L_tmpnam. При каждом вызове tmpnam генерируется новое имя; при этом гарантируется не более TMP_MAX различных имен за один сеанс работы программы. Заметим, что tmpnam создает имя, а не файл.

int setvbuf(FILE *stream, char *buf, int mode, size_t size)

setvbuf управляет буферизацией потока; к ней следует обращаться прежде, чем будет выполняться чтение, запись или какая-либо другая операция. mode со значением _IOFBF вызывает полную буферизацию, с _IOLBF - "построчную" буферизацию текстового файла, а mode со значением _IONBF отменяет всякую буферизацию. Если параметр buf не есть NULL, то его значение - указатель на буфер, в противном случае под буфер будет запрашиваться память. Параметр size задает размер буфера. Функция setvbuf в случае ошибки выдает ненулевое значение.

void setbuf(FILE *stream, char *buf)

Если buf есть NULL, то для потока stream буферизация выключается. В противном случае вызов setbuf приведет к тем же действиям, что и вызов (void) setvbuf (stream, buf, _IOFBF, BUFSIZ).

В 1.2. Форматный вывод

Функции printf осуществляют вывод информации по формату.

int fprintf(FILE *stream, const char *format, ...)

fprintf преобразует и пишет вывод в поток stream под управлением format. Возвращаемое значение - число записанных символов или, в случае ошибки, отрицательное значение.

Форматная строка содержит два вида объектов: обычные символы, копируемые в выводной поток, и спецификации преобразования, которые вызывают преобразование и печать остальных аргументов в том порядке, как они перечислены. Каждая спецификация преобразования начинается с % и заканчивается символом-спецификатором преобразования. Между % и символом-спецификатором в порядке, в котором они здесь перечислены, могут быть расположены следующие элементы информации:

- Флаги (в любом порядке), модифицирующие спецификацию:

  - - указывает на то, что преобразованный аргумент должен быть прижат к левому краю поля;

  + - предписывает печатать число всегда со знаком;

  пробел - если первый символ - не знак, то числу должен предшествовать пробел;

  0 - указывает, что числа должны дополняться слева нулями до всей ширины поля;

  # - указывает на одну из следующих форм вывода: для о первой цифрой должен быть 0; для x или X ненулевому результату должны предшествовать 0x или 0X; для e, E, f, g и G вывод должен

обязательно содержать десятичную точку; для g и 6 завершающие нули не отбрасываются.

- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет напечатан в поле, размер которого не меньше указанной ширины, а если потребуется, в поле большего размера. Если число символов преобразованного аргумента меньше ширины поля, то поле будет дополнено слева (или справа, если число прижимается к левому краю). Обычно поле дополняется пробелами (или нулями, если присутствует флаг дополнения нулями).
- Точка, отделяющая указатель ширины поля от указателя точности.
- Число, задающее точность, которое специфицирует максимальное количество символов, печатаемых из строки, или количество цифр после десятичной точки в преобразованиях e, E или f, или количество значащих цифр для g или G-преобразования, или минимальное количество цифр при печати целого (до необходимой ширины поля число дополняется слева нулями).
- Модификаторы h, 1 (буква ell) или L. "h" указывает на то, что соответствующий аргумент должен печататься как short или unsigned short; "1" сообщает, что аргумент имеет тип long или unsigned long; "L" информирует, что аргумент принадлежит типу long double.

Ширина, или точность, или обе эти характеристики могут быть специфицированы с помощью *; в этом случае необходимое число "извлекается" из следующего аргумента, который должен иметь тип int (в случае двух звездочек используются два аргумента).

Символы-спецификаторы и разъяснение их смысла приведены в таблице В-1. Если за % нет правильного символа-спецификатора, результат не определен.

int printf(const char *format, ...)

printf(...) полностью эквивалентна fprintf(stdout, ...).

int sprintf(char *s, const char *format, ...)

sprintf действует так же, как и printf, только вывод осуществляет в строку s, завершая ее символом '\0'. Строка s должна быть достаточно большой, чтобы вмещать результат вывода. Возвращает количество записанных символов, в число которых символ '\0' не входит.

int vprintf (const char *format, va_list arg)
int vfprintf (FILE *stream, const char *format, va_list arg)
int vsprintf (char *s, const char *format, va_list arg)

Функции vprintf, vfprintf и vsprintf эквивалентны соответствующим printf-функциям с той лишь разницей, что переменный список аргументов представлен параметром arg, инициализированным макросом va_start и, возможно, вызовами va_arg (см. в В7 описание <stdarg.h>).

Таблица В-1. **Преобразования** printf

Символ	Тип аргумента; вид печати
d, i	int; знаковая десятичная запись
o	int; беззнаковая восьмеричная запись (без 0 слева)
x, X	unsigned int; беззнаковая шестнадцатеричная запись (без Ox или OX слева), в качестве цифр от 10 до 15 используются abcdef для x и ABCDEF для X
u	int; беззнаковое десятичное целое
c	int; единичный символ после преобразования в unsigned char.
s	char *; символы строки печатаются, пока не встретится '\0' или не исчерпается количество символов, указанное точностью
f	double; десятичная запись вида [-]*mmm.ddd,* где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки
e, E	double; десятичная запись вида [-]*m.dddddd*e±*xx* или запись вида [-]*m.dddddd*E±*xx,* где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки
g, G	double; используется %e и %E, если порядок меньше -4 или больше или равен точности; в противном случае используется %f. Завершающие нули и точка в конце не печатаются
p	void *; печатает в виде указателя (представление зависит от реализации)
n	int *; число символов, напечатанных к данному моменту данным вызовом printf, *записывается в* аргумент. Никакие другие аргументы не преобразуются
%	никакие аргументы не преобразуются; печатается %

В 1.3. Форматный ввод

Функции scanf имеют дело с форматным преобразованием при вводе.

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf читает данные из потока stream под управлением format и преобразованные величины присваивает по порядку аргументам, *каждый из которых должен быть указателем.* Завершает работу, если исчерпался формат. Выдает EOF

по исчерпании файла или перед любым преобразованием, если возникла ошибка; в остальных *случаях* функция возвращает количество преобразованных и введенных элементов.

Форматная строка обычно содержит спецификации преобразования, которые используются для управления вводом. В форматную строку могут входить:

- пробелы и табуляции, которые игнорируются;
- обычные символы (кроме %), которые ожидаются в потоке ввода среди символов, отличных от символов-разделителей;
- спецификации преобразования, состоящие из %; необязательного знака *, подавляющего присваивание; необязательного числа, специфицирующего максимальную ширину поля; необязательных h, l или L, указывающих размер присваиваемого значения, и символа-спецификатора преобразования.

Спецификация преобразования определяет преобразование следующего поля ввода. Обычно результат размещается в переменной, на которую указывает соответствующий аргумент. Однако если присваивание подавляется с помощью знака *, как, например, в %*s, то поле ввода просто пропускается, и никакого присваивания не происходит. Поле ввода определяется как строка символов, отличных от символов-разделителей; при этом ввод строки прекращается при выполнении любого из двух условий: если встретился символ-разделитель или если ширина поля (в случае, когда она указана) исчерпана. Из этого следует, что при переходе к следующему полю scanf может "перешагивать" через границы строк, поскольку символ новой строки является символом-разделителем. (Под символами-разделителями понимаются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и смены страницы.)

Символ-спецификатор указывает на способ интерпретации поля ввода. Соответствующий аргумент должен быть указателем. Список допустимых символов-спецификаторов приводится в таблице В-2.

Символам-спецификаторам d, i, n, o, u и x может предшествовать h, если аргумент есть указатель на short (а не int) или l (буква ell), если аргумент есть указатель на long. Символам-спецификаторам e, f и g может предшествовать l, если аргумент - указатель на double (а не float), или L, если аргумент - указатель на long double.

`int scanf (const char *format, ...)`

scanf(...) делает то же, что и fscanf (stdin, ...).

`int sscanf (const char *s, const char *format, ...)`

sscanf(s, ...) делает то же, что и scanf(...), только ввод символов осуществляет из строки s.

Таблица В-2. **Преобразования** `scanf`

Символ	**Данные на вводе; тип аргумента**
d	десятичное целое; int *
i	целое; int *. Целое может быть восьмеричным (с нулем слева) или шестнадцатеричным (с 0x или 0X слева)
o	восьмеричное целое (с нулем слева или без него); int *
u	беззнаковое десятичное целое; unsigned int *
x	шестнадцатеричное целое (с 0x или 0X слева или без них); int *
c	символы; char *. Символы ввода размещаются в указанном массиве в количестве, заданном шириной поля; по умолчанию это количество равно 1. Символ '\0' не добавляется. Символы-разделители здесь рассматриваются как обычные символы и поступают в аргумент. Чтобы прочесть следующий символ-разделитель, используйте %1s
s	строка символов, отличных от символов-разделителей (записывается без кавычек); char *, указывающий на массив размера достаточного, чтобы вместить строку и добавляемый к ней символ '\0'
e, f, g	число с плавающей точкой; float *. Формат ввода для float состоит из необязательного знака, строки цифр, возможно с десятичной точкой, и необязательного порядка, состоящего из E или e и целого, возможно со знаком
p	значение указателя в виде, в котором printf ("%p") его напечатает; void *
n	записывает в аргумент число символов, прочитанных к этому моменту в этом вызове; int *. Никакого чтения ввода не происходит. Счетчик числа введенных элементов не увеличивается
[...]	выбирает из ввода самую длинную непустую строку, состоящую из символов, заданных в квадратных скобках; char *. В конец строки добавляется '\0'. Спецификатор вида []...] включает] в задаваемое множество символов
[^...]	выбирает из ввода самую длинную непустую строку, состоящую из символов, *не входящих* в заданное в скобках множество. В конец добавляется '\0'. Спецификатор вида [^]...] включает] в задаваемое множество символов
%	обычный символ %; присваивание не делается

В 1.4. Функции ввода-вывода символов

int fgetc(FILE *stream)

fgetc возвращает следующий символ из потока stream в виде unsigned char (переведенную в int) или EOF, если исчерпан файл или обнаружена ошибка.

char *fgets(char *s, int n, FILE *stream)

fgets читает не более n-1 символов в массив s, прекращая чтение, если встретился символ новой строки, который включается в массив; кроме того, записывает в массив '\0'. Функция fgets возвращает s или, если исчерпан файл или обнаружена ошибка, NULL.

int fputc(int c, FILE *stream)

fputc пишет символ c (переведенный в unsigned char) в stream. Возвращает записанный символ или EOF в случае ошибки.

int fputs(const char *s, FILE *stream)

fputs пишет строку s (которая может не иметь '\n') в stream; возвращает неотрицательное целое или EOF в случае ошибки.

int getc(FILE *stream)

getc делает то же, что и fgetc, но в отличие от последней, если она - макрос, stream может браться более одного раза.

int getchar(void)

getchar() делает то же, что getc(stdin).

char *gets(char *s)

gets читает следующую строку ввода в массив s, заменяя символ новой строки на '\0'. Возвращает s или, если исчерпан файл или обнаружена ошибка, NULL.

int putc(int c, FILE *stream)

putc делает то же, что и fputc, но в отличие от последней, если putc - макрос, значение stream может браться более одного раза.

int putchar(int c)

putchar(c) делает то же, что putc(c, stdout).

int puts(const char *s)

puts пишет строку s и символ новой строки в stdout. Возвращает EOF в случае ошибки, или неотрицательное значение, если запись прошла нормально.

int ungetc(int c, FILE *stream)

ungetc отправляет символ c (переведенный в unsigned char) обратно в stream;

при следующем чтении из stream он будет получен снова. Для каждого потока вернуть можно не более одного символа. Нельзя возвращать EOF. В качестве результата ungetc выдает отправленный назад символ или, в случае ошибки, EOF.

В 1.5. Функции прямого ввода-вывода

size_t fread(void *ptr, size_t size, size_t nobj, FILE *strem)

fread читает из потока stream в массив ptr не более nobj объектов размера size. Она возвращает количество прочитанных объектов, которое может быть меньше заявленного. Для индикации состояния после чтения следует использовать feof и ferror.

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

fwrite пишет из массива ptr в stream nobj объектов размера size; возвращает число записанных объектов, которое в случае ошибки меньше nobj.

В1.6. Функции позиционирования файла

int fseek(FILE *stream, long offset, int origin)

fseek устанавливает позицию для stream; последующее чтение или запись будет производиться с этой позиции. В случае бинарного файла позиция устанавливается со смещением offset - относительно начала, если origin равен SEEK_SET; относительно текущей позиции, если origin равен SEEK_CUR; и относительно конца файла, если origin равен SEEK_END. Для текстового файла offset должен быть нулем или значением, полученным с помощью вызова функции ftell. При работе с текстовым файлом origin всегда должен быть равен SEEK_SET.

long ftell(FILE *stream)

ftell возвращает текущую позицию потока stream или -1L, в случае ошибки.

void rewind(FILE *stream)

rewind (fp) делает то же, что и fseek(fp, OL, SEEK_SET); clearerr(fp).

int fgetpos(FILE *stream, fpos_t *ptr)

fgetpos записывает текущую позицию потока stream в *ptr для последующего использования ее в fsetpos. Тип fpos_t позволяет хранить такого рода значения. В случае ошибки fgetpos возвращает ненулевое значение.

int fsetpos(FILE *stream, const fpos_t *ptr)

fsetpos устанавливает позицию в stream, читая ее из *ptr, куда она была записана ранее с помощью fgetpos. В случае ошибки fsetpos возвращает ненулевое значение.

B 1.7. Функции обработки ошибок

Многие функции библиотеки в случае ошибки или конца файла устанавливают индикаторы состояния. Эти индикаторы можно проверять и изменять. Кроме того, целое выражение errno (объявленное в <errno.h>) может содержать номер ошибки, который дает дополнительную информацию о последней из обнаруженных ошибок.

`void clearerr(FILE *stream)`

clearerr очищает индикаторы конца файла и ошибки потока stream.

`int feof(FILE *stream)`

feof возвращает ненулевое значение, если для потока stream установлен индикатор конца файла.

`int ferror(FILE *stream)`

ferror возвращает ненулевое значение, если для потока stream установлен индикатор ошибки.

`void perror(const char *s)`

perror(s) печатает s и зависимое от реализации сообщение об ошибке, соответствующее целому значению в errno, т. е. делает то же, что и обращение к функции fprintf вида

```
fprintf(stderr, "%s: %s\n", s, "сообщение об ошибке")
```

См. strerror в параграфе B3.

B 2. Проверки класса символа: <ctype.h>

Заголовочный файл <ctype.h> объявляет функции, предназначенные для проверок символов. Аргумент каждой из них имеет тип int и должен либо представлять собой EOF, либо быть значением unsigned char, приведенным к int; возвращаемое значение тоже имеет тип int. Функции возвращают ненулевое значение ("истина"), когда аргумент c удовлетворяет описанному условию или принадлежит указанному классу символов, и нуль в противном случае.

isalnum(c)	- isalpha(c) или isdigit(c) есть истина
isalpha(c)	- isupper(c) или islower(c) есть истина
iscntrl(c)	- управляющий символ
isdigit(c)	- десятичная цифра
isgraph(c)	- печатаемый символ кроме пробела
islower(c)	- буква нижнего регистра
isprint(c)	- печатаемый символ, включая пробел
ispunct(c)	- печатаемый символ кроме пробела, буквы или цифры

isspace(c)	- пробел, смена страницы, новая строка, возврат каретки, табуляция, вертикальная табуляция
isupper(c)	- буква верхнего регистра
isxdigit(c)	- шестнадцатеричная цифра

В наборе семибитовых ASCII-символов печатаемые символы находятся в диапазоне от 0x20 (' ') до 0x7E ('~'); управляющие символы - от 0 (NUL) до 0x1F (US) и 0x7F (DEL).

Помимо перечисленных есть две функции, приводящие буквы к одному из регистров:

int tolower(int c)	- переводит c на нижний регистр;
int toupper(int c)	- переводит c на верхний регистр.

Если c - буква на верхнем регистре, то tolower(c) выдаст эту букву на нижнем регистре; в противном случае она вернет c. Если c - буква на нижнем регистре, то toupper(c) выдаст эту букву на верхнем регистре; в противном случае она вернет c.

В 3. Функции, оперирующие со строками: <string.h>

Имеются две группы функций, оперирующих со строками. Они определены в заголовочном файле <string.h>. Имена функций первой группы начинаются с букв str, второй - с mem. Если копирование имеет дело с объектами, перекрывающимися по памяти, то, за исключением memmove, поведение функций не определено. Функции сравнения рассматривают аргументы как массивы элементов типа unsigned char.

В таблице на с. 321 переменные s и t принадлежат типу char *, cs и ct - типу const char *, n - типу size_t, а c - значение типа int, приведенное к типу char.

Последовательные вызовы strtok разбивают строку s на лексемы. Ограничителем лексемы служит любой символ из строки ct. В первом вызове указатель s не равен NULL. Функция находит в строке s первую лексему, состоящую из символов, не входящих в ct; ее работа заканчивается тем, что поверх следующего символа пишется '\0' и возвращается указатель на лексему. Каждый последующий вызов, в котором указатель s равен NULL, возвращает указатель на следующую лексему, которую функция будет искать сразу за концом предыдущей. Функция strtok возвращает NULL, если далее никакой лексемы не обнаружено. Параметр ct от вызова к вызову может варьироваться.

Функции mem... предназначены для манипулирования с объектами как с массивами символов; их назначение - получить интерфейсы к эффективным

char*strcpy(s,ct)	копирует строку ct в строку s, включая '\0'; возвращает s
char *strncpy(s,ct, n)	копирует не более n символов строки ct в s; возвращает s. Дополняет результат символами '\0', если символов в ct меньше n
char *strcat(s,ct)	приписывает ct к s; возвращает s
char *strncat(s,ct, n)	приписывает не более n символов ct к s, завершая s символом '\0'; возвращает s
char strcmp(cs,st)	сравнивает cs и ct; возвращает <0, если cs<ct, 0, если cs==ct, и >0, если cs>ct[1]
char strncmp(cs,ct)	сравнивает не более n символов cs и ct; возвращает <0, если cs<ct, 0, если cs==ct, и >0, если cs>ct
char *strchr(cs,c)	возвращает указатель на первое вхождение c в cs или, если такового не оказалось, NULL
char *strrchr(cs,c)	возвращает указатель на последнее вхождение c в cs или, если такового не оказалось, NULL
size_t strspn(cs,ct)	возвращает длину начального сегмента cs, состоящего из символов, входящих в строку ct
size_t strcspn(cs,ct)	возвращает длину начального сегмента cs, состоящего из символов, не входящих в строку ct
char *strpbrk(cs,ct)	возвращает указатель в cs на первый символ, который совпал с одним из символов, входящих в ct, или, если такового не оказалось, NULL
char *strstr(cs,ct)	возвращает указатель на первое вхождение ct в cs или, если такового не оказалось, NULL
size_t strlen(cs)	возвращает длину cs
char * strerror(n)	возвращает указатель на зависящую от реализации строку, соответствующую номеру ошибки n
char * strtok(s,ct)	strtok ищет в s лексему, ограниченную символами из ct; более подробное описание этой функции см. ниже

[1] Здесь и ниже под такими выражениями как cs<ct не следует понимать арифметическое сравнение указателей. Подразумевается лексикографическое сравнение, т. е. cs меньше (больше) ct, если первый несовпавший элемент в cs арифметически меньше (больше) соответствующего элемента из ct. — *Примеч. ред.*

I! Зак. 1116

программам. В приведенной ниже таблице s и t принадлежат типу void *; cs и ct - типу const void *; n - типу size_t; а c имеет значение типа int, приведенное к типу char.

void *memcpy(s,ct,n)	копирует n символов из ct в s и возвращает s
void *memmove(s, ct, n)	делает то же самое, что и memcpy, но работает и в случае "перекрывающихся" объектов.
int memcmp(cs, ct, n)	сравнивает первые n символов cs и ct; выдает тот же результат, что и функция strcmp
void *memchr(cs,c,n)	возвращает указатель на первое вхождение символа c в cs или, если среди первых n символов c не встретилось, NULL
void *memset(s,c, n)	размещает символ c в первых n позициях строки s и возвращает s

В 4. Математические функции: <math.h>

В заголовочном файле <math.h> описываются математические функции и определяются макросы.

Макросы EDOM и ERANGE (находящиеся в <errno.h>) задают отличные от нуля целочисленные константы, используемые для фиксации ошибки области и ошибки диапазона; HUGE_VAL определена как положительное значение типа double. *Ошибка области* возникает, если аргумент выходит за область значений, для которой определена функция. Фиксация ошибки области осуществляется присвоением errno значения EDOM; возвращаемое значение зависит от реализации. *Ошибка диапазона* возникает тогда, когда результат функции не может быть представлен в виде double. В случае переполнения функция возвращает HUGE_VAL с правильным знаком и в errno устанавливается значение ERANGE. Если результат оказывается меньше, чем возможно представить данным типом, функция возвращает нуль, а устанавливается ли в этом случае в errno ERANGE, зависит от реализации. Далее x и y имеют тип double, n - тип int, и все функции возвращают значения типа double. Углы в тригонометрических функциях задаются в радианах.

sin(x)	синус x
cos(x)	косинус x
tan(x)	тангенс x
asin(x)	арксинус x в диапазоне $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
acos(x)	арккосинус x в диапазоне $[0, \pi]$, $x \in [-1, 1]$

atan(x)	арктангенс x в диапазоне [-π/2, я/2]
atan2(y,x)	арктангенс y/x в диапазоне [-π, л]
sinh(x)	гиперболический синус x
cosh(x)	гиперболический косинус x
tanh(x)	гиперболический тангенс x
exp(x)	экспоненциальная функция e^x
log(x)	натуральный логарифм $\ln(x)$, $x >$ O
loglO(x)	десятичный логарифм $\log_{10}(x) x > 0$
pow(x,y)	x^y. Ошибка области, если x = O и $y \leqslant$ O или $x < 0$ и y - не целое
sqrt(x)	$\sqrt{x}$, $x \geqslant 0$
ceil(x)	наименьшее целое в виде double, которое не меньше ж
floor(x)	наибольшее целое в виде double, которое не больше x
fabs(x)	абсолютное значение $\|x$ \
ldexp(x,n)	$x \cdot 2^n$
frexp(x, int *exp)	разбивает x на два сомножителя, первый из которых - нормализованная дробь в интервале [1/2, 1), которая возвращается, а второй - степень двойки, эта степень запоминается в *exp. Если x - нуль, то обе части результата равны нулю
modf(x, double *ip)	разбивается на целую и дробную части, обе имеют тот же знак, что и x. Целая часть запоминается в *ip, дробная часть выдается как результат
fmod(x,y)	остаток от деления x на y в виде числа с плавающей точкой. Знак результата совпадает со знаком x. Если y равен нулю, результат зависит от реализации

В 5. Функции общего назначения: <stdlib. h>

Заголовочный файл <stdlib.h> объявляет функции, предназначенные для преобразования чисел, запроса памяти и других задач.

double atof(const char *s)

atof **переводит** s **в** double; **эквивалентна** strtod(s, (char**) NULL).

int atoi(const char *s)

переводит s **в** int; **эквивалентна** (int)strtol(s. (char**)NULL, 10).

int atol(const char *s)

переводит s в long; эквивалентна strtol(s, (char**) NULL, 10).

double strtod(const char *s, char **endp)

strtod преобразует первые символы строки s в double, игнорируя начальные символы-разделители; запоминает указатель на непреобразованный конец в *endp (если endp не NULL). при переполнении она выдает HUGE_VAL с соответствующим знаком, в случае, если результат оказывается меньше, чем возможно представить данным типом, возвращается 0; в обоих случаях в errno устанавливается ERANGE.

long strtol(const char *s, char **endp, int base)

strtol преобразует первые символы строки s в long, игнорируя начальные символы-разделители; запоминает указатель на непреобразованный конец в *endp (если endp не NULL). Если base находится в диапазоне от *2* до 36, то преобразование делается в предположении, что на входе - запись числа по основанию base. Если base равно нулю, то основанием числа считается 8, 10 или 16; число, начинающееся с цифры 0, считается восьмеричным, а с Ox или OX - шестнадцатеричным. Цифры от 10 до base-1 записываются начальными буквами латинского алфавита в любом регистре. При основании, равном 16, в начале числа разрешается помещать Ox или OX. В случае переполнения функция возвращает LONG_MAX или LONG_MIN (в зависимости от знака), а в еггпо устанавливается ERANGE.

unsigned long strtoul(const char *s, char **endp, int base) .

strtoul работает так же, как и strtol, с той лишь разницей, что возвращает результат типа unsigned long, а в случае переполнения - ULONG_MAX.

int rand(void)

rand **выдает псевдослучайное число в диапазоне от 0 до** RAND_MAX; RAND_MAX **не меньше 32767.**

void srand(unsigned int seed)

srand использует seed в качестве семени для новой последовательности псевдослучайных чисел. Изначально параметр seed равен 1.

void *calloc(size_t nobj, size_t size)

calloc возвращает указатель на место в памяти, отведенное для массива nobj объектов, каждый из которых размера size, или, если памяти запрашиваемого объема нет, NULL. Выделенная область памяти обнуляется.

void *malloc(size_t size)

malloc возвращает указатель на место в памяти для объекта размера size или, если памяти запрашиваемого объема нет, NULL. Выделенная область памяти не инициализируется.

void *realloc(void *p, size_t size)

realloc заменяет на size размер объекта, на который указывает p. Для части, размер которой равен наименьшему из старого и нового размеров, содержимое не изменяется. Если новый размер больше старого, дополнительное пространство не инициализируется, realloc возвращает указатель на новое место памяти или, если требования не могут быть удовлетворены, NULL (*p при этом не изменяется).

void free(void *p)

free освобождает область памяти, на которую указывает p; эта функция ничего не делает, если p равно NULL. В p должен стоять указатель на область памяти, ранее выделенную одной из функций: calloc, malloc или realloc.

void abort(void *p)

abort вызывает аварийное завершение программы, ее действия эквивалентны вызову raise(SIGABRT).

void exit(int status)

exit вызывает нормальное завершение программы. Функции, зарегистрированные с помощью atexit, выполняются в порядке, обратном их регистрации. Производится опорожнение буферов открытых файлов, открытые потоки закрываются, и управление возвращается в среду, из которой был произведен запуск программы. Значение status, передаваемое в среду, зависит от реализации, однако при успешном завершении программы принято передавать нуль. Можно также использовать значения EXIT_SUCCESS (в случае успешного завершения) и EXIT_FAILURE (в случае ошибки).

int atexit(void (*fcn)(void))

atexit регистрирует fcn в качестве функции, которая будет вызываться при нормальном завершении программы; возвращает ненулевое значение, если регистрация не может быть выполнена.

int system(const char *s)

system передает строку s операционной среде для выполнения. Если s есть NULL и существует командный процессор, то system возвращает ненулевое значение. Если s не NULL, то возвращаемое значение зависит от реализации.

char *getenv(const char *name)

getenv возвращает строку среды, связанную с name, или, если никакой строки не существует, NULL. Детали зависят от реализации.

```
void *bsearch(const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *datum))
```

bsearch ищет среди base[0]...base[n-1] элемент с подходящим ключом *key.

Функция cmp должна сравнивать первый аргумент (ключ поиска) со своим вторым аргументом (значением ключа в таблице) и в зависимости от результата сравнения выдавать отрицательное число, нуль или положительное значение. Элементы массива base должны быть упорядочены в возрастающем порядке, bsearch возвращает указатель на элемент с подходящим ключом или, если такого не оказалось, NULL.

void qsort(void *base, size_t n, size_t size,
                int (*cmp)(const void *, const void *))

qsort сортирует массив base[0]...base[n-1] объектов размера size в возрастающем порядке. Функция сравнения стр - такая же, как и в bsearch.

int abs(int n)

abs возвращает абсолютное значение аргумента типа int.

long labs(long n)

labs возвращает абсолютное значение аргумента типа long.

div_t div(int num, int denom)

div вычисляет частное и остаток от деления num на denom. Результаты типа int запоминаются в элементах quot и rem структуры div_t.

ldiv_t ldiv(long num, long denom)

ldiv вычисляет частное и остаток от деления num на denom. Результаты типа long запоминаются в элементах quot и rem структуры ldiv_t.

В 6. Диагностика: <assert.h>

Макрос assert используется для включения в программу диагностических сообщений.

void assert (int *выражение)*

Если *выражение* имеет значение нуль, то

assert *(выражение)*

напечатает в stderr сообщение следующего вида:

Assertion failed: *выражение,* file *имя-файла,* line *nnn*

после чего будет вызвана функция abort, которая завершит вычисления. Имя исходного файла и номер строки будут взяты из макросов __FILE__ и __LINE__.

Если в момент включения файла <assert.h> было определено имя NDEBUG, то макрос assert игнорируется.

В 7. Списки аргументов переменной длины: <stdarg.h>

Заголовочный файл <stdarg. h> предоставляет средства для перебора аргументов функции, количество и типы которых заранее не известны.

Пусть *посларг* - последний именованный параметр функции f с переменным числом аргументов. Внутри f объявляется переменная ap типа va_list, предназначенная для хранения указателя на очередной аргумент:

```
va_list ap;
```

Прежде чем будет возможен доступ к безымянным аргументам, необходимо один раз инициализировать ap, обратившись к макросу va_start:

```
va_start(va_list ap, посларг);
```

С этого момента каждое обращение к макросу:

```
тип va_arg(va_list ap, тип);
```

будет давать значение очередного безымянного аргумента указанного типа, и каждое такое обращение будет вызывать автоматическое приращение указателя ap, чтобы последний указывал на следующий аргумент. Один раз после перебора аргументов, но до выхода из f необходимо обратиться к макросу

```
void va_end(va_list ap);
```

В 8. Дальние переходы: <setjmp. h>

Объявления в <setjmp.h> предоставляют способ отклониться от обычной последовательности "вызов - возврат"; типичная ситуация - необходимость вернуться из "глубоко вложенного" вызова функции на верхний уровень, минуя промежуточные возвраты.

int setjmp(jmp_buf env)

Макрос setjmp сохраняет текущую информацию о вызовах в env для последующего ее использования в longjmp. Возвращает нуль, если возврат осуществляется непосредственно из setjmp, и не нуль, если - от последующего вызова longjmp. Обращение к setjmp возможно только в определенных контекстах; в основном это проверки в if, switch и циклах, причем только в простых выражениях отношения.

```
if (setjmp() == 0)
   /* после прямого возврата */
else
   /* после возврата из longjmp */
```

void longjmp(jmp_buf env, int val)

longjmp восстанавливает информацию, сохраненную в самом последнем вызове setjmp, по информации из env; выполнение программы возобновляется, как если бы функция setjmp только что отработала и вернула ненулевое значение val. Результат будет непредсказуемым, если в момент обращения к longjmp функция, содержащая вызов setjmp, уже "отработала" и осуществила возврат. Доступные ей объекты имеют те значения, которые они имели в момент обращения к longjmp; setjmp не сохраняет значений.

В 9. Сигналы: <signal.h>

Заголовочный файл <signal.h> предоставляет средства для обработки исключительных ситуаций, возникающих во время выполнения программы, таких как прерывание, вызванное внешним источником или ошибкой в вычислениях.

void (*signal(int sig, void (*handler)(int)))(int)

signal устанавливает, как будут обрабатываться последующие сигналы. Если параметр handler имеет значение SIG_DFL, то используется зависимая от реализации "обработка по умолчанию"; если значение handler равно SIG_IGN, то сигнал игнорируется; в остальных случаях будет выполнено обращение к функции, на которую указывает handler с типом сигнала в качестве аргумента. В число допустимых видов сигналов входят:

SIGABRT	- аварийное завершение, например от abort;
SIGFPE	- арифметическая ошибка: деление на 0 или переполнение;
SIGILL	- неверный код функции (недопустимая команда);
SIGINT	- запрос на взаимодействие, например прерывание;
SIGSEGV	- неверный доступ к памяти, например выход за границы;
SIGTERM	- требование завершения, посланное в программу.

signal возвращает предыдущее значение handler в случае специфицированного сигнала, или SIGERR в случае возникновения ошибки.

Когда в дальнейшем появляется сигнал sig, сначала восстанавливается готовность поведения "по умолчанию", после чего вызывается функция, заданная в параметре handler, т. е. как бы выполняется вызов (*handler)(sig). Если функция handler вернет управление назад, то вычисления возобновятся с того места, где застал программу пришедший сигнал.

Начальное состояние сигналов зависит от реализации.

int raise(int sig)

raise посылает в программу сигнал sig. В случае неудачи возвращает ненулевое значение.

В 10. Функции даты и времени: <time.h>

Заголовочный файл <time.h> объявляет типы и функции, связанные с датой и временем. Некоторые функции имеют дело с *местным временем*, которое может отличаться от календарного, например в связи с зонированием времени. Типы clock_t и time_t - арифметические типы для представления времени, а struct tm содержит компоненты календарного времени:

```
int tm_sec;   - секунды от начала минуты (0,61);
int tm_min;   - минуты от начала часа (0,59);
int tm_hour;  - часы от полуночи (0,23);
int tm_mday;  - число месяца (1,31);
int tm_mon;   - месяцы с января (0,11);
int tm_year;  - годы с 1900;
int tm_wday;  - дни с воскресенья (0,6);
int tm_yday;  - дни с 1 января (0,365);
int tm_isdst; - признак летнего времени.
```

Значение tm_isdst - положительное, если время приходится на сезон, когда время суток сдвинуто на 1 час вперед, нуль в противном случае и отрицательное, если информация не доступна.

clock_t clock(void)

clock возвращает время, фиксируемое процессором от начала выполнения программы, или -1, если оно не известно. Для выражения этого времени в секундах применяется формула clock()/CLOCKS_PER_SEC.

time_t time(time_t *tp)

time возвращает текущее календарное время (т. е. время, прошедшее после определенной даты, - обычно после 0 ч 00 мин 00 с GMT 1-го января 1970 г. - *примеч. ред.*) или -1, если время не известно. Если tp не равно NULL, то возвращаемое значение записывается и в *tp.

double difftime(time_t time2, time_t time1)

difftime возвращает разность time2-time1, выраженную в секундах.

time_t mktime(struct tm «tp)

mktime преобразует местное время, заданное структурой *tp, в календарное, выдавая его в том же виде, что и функция time. Компоненты будут иметь значения в указанных диапазонах. Функция mktime возвращает календарное время или -1, если оно не представимо.

Следующие четыре функции возвращают указатели на статические объекты, каждый из которых может быть изменен другими вызовами.

char *asctime(const struct tm *tp)

asctime переводит время в структуре *tp в строку вида

Sun Jan 3 15:14:13 1988\n\0

char *ctime(const time_t *tp)

ctime переводит календарное время в местное, что эквивалентно выполнению

asctime(localtime(tp))

struct tm *gmtime(const time_t *tp)

gmtime переводит календарное время во Всемирное координированное время (Coordinated Universal Time - UTC). Выдает NULL, если UTC не известно. Имя этой **функции**, gmtime, происходит от Greenwich Mean Time (среднее время по Гринвичскому меридиану).

struct tm *localtime(const time_t *tp)

localtime переводит календарное время *tp в местное.

size_t strftime(char *s, size_t smax, const char *fmt,
const struct tm *tp)

strftime форматирует информацию о дате и времени из *tp в строку s согласно формату fmt, который имеет много общих черт с форматом, задаваемым в функции printf. Обычные символы (включая и завершающий символ '\0') копируются в *s*. Каждая пара, состоящая из % и буквы, заменяется, как показано ниже, с использованием значений по форме, соответствующей местным традициям. В s размещается не более smax символов, strftime возвращает число символов без учета '\0' или нуль, если число сгенерированных символов больше smax.

%a	сокращенное название дня недели
%A	полное название дня недели
%b	сокращенное название месяца
%B	полное название месяца.
%c	местное представление даты и времени
%d	день месяца (01-31)
%H	час (24-часовое время) (00-23)
%I	час (12-часовое время) (01-12)
%j	день от начала года (001-366)
%т	месяц (01-12)
%M	минута (00-59)
%p	местное представление AM или PM (до или после полудня)
%S	секунда (00-61)
%U	неделя от начала года (считая, что воскресенье - 1-й день недели) (00-53)
%w	день недели (0-6, номер воскресенья - 0)
%W	неделя от начала года (считая, что понедельник - 1-й день недели) (00-53)

%x	местное представление даты
%X	местное представление времени
%y	год без указания века (00-99)
%Y	год с указанием века
%Z	название временной зоны, если она есть
%%	%

В 11. Зависящие от реализации пределы: <limits.h> и <float.h>

Заголовочный файл <limits.h> определяет константы для размеров целочисленных типов. Ниже перечислены минимальные приемлемые величины, но в конкретных реализациях могут использоваться и большие значения.

CHAR_BIT	8	битов в значении char
SCHAR_MAX	UCHAR_MAX *или* SCHAR_MAX	максимальное значение char
CHAR_MIN	0 *или* SCHAR_MIN	минимальное значение char
INT_MAX	+32767	максимальное значение int
INT_MIN	-32767	минимальное значение int
LONG_MAX	+2147463647	максимальное значение long
LONG_MIN	-2147483647	минимальное значение long
SCHAR_MAX	+127	максимальное значение signed char
SCHAR_MIN	-127	минимальное значение signed char
SHRT_MAX	+32767	максимальное значение short
SHRT_MIN	-32767	минимальное значение short
UCHAR_MAX	255	максимальное значение unsigned char
UINT_MAX	65535	максимальное значение unsigned int
ULONG_MAX	4294967295	максимальное значение unsigned long
USHRT_MAX	65535	максимальное значение unsigned short

Имена, приведенные в следующей таблице, взяты из <float.h> и являются константами, имеющими отношение к арифметике с плавающей точкой. Значения (если они есть) представляют собой минимальные значения для соответствующих величин. В каждой реализации устанавливаются свои значения.

FLT_RADIX	2	основание для представления порядка, например: 2, 16
FLT_ROUNDS		способ округления при сложении чисел с плавающей точкой

FLT_DIG	6	количество верных десятичных цифр
FLT_EPSILON	1E-5	минимальное x, такое, что $1.0 + x \neq 1.0$
FLT_MANT_DIG		количество цифр по основанию FLT_RADIX в мантиссе
FLT_MAX	1E+37	максимальное число с плавающей точкой
FLT_MAX_EXP		максимальное n, такое, что FLT_RADIX^n - 1 представимо
FLT_MIN	1E-37	минимальное нормализованное число с плавающей точкой
FLT_MIN_EXP		минимальное n, такое, что 10^n представимо в виде нормализованного числа
DBL_DIG	10	количество верных десятичных цифр для типа double
DBL_EPSILON	1E-9	минимальное x, такое, что $1.0 + x \neq 1.0$, где x принадлежит типу double
DBL_MANT_DIG		количество цифр по основанию FLT_RADIX в мантиссе для чисел типа double
DBL_MAX	1E+37	максимальное число с плавающей точкой типа double
DBL_MAX_EXP		максимальное n, такое, что FLT_RADIX^n -1 представимо в виде числа типа double
DBL_MIN	1E-37	минимальное нормализованное число с плавающей точкой типа double
DBL_MIN_EXP		минимальное n, такое, что 10^n представимо в виде нормализованного числа типа double

Перечень изменений

С момента публикации первого издания этой книги определение языка Си претерпело изменения. Почти все нововведения - это расширения исходной версии языка, выполненные так, чтобы сохранилась совместимость с существующими программами; некоторые изменения касаются устранения двусмысленностей первоначального описания, а некоторые представляют собой модификации, привнесенные существующей практикой. Многие из новых возможностей, впоследствии принятые другими разработчиками Си-компиляторов, были первоначально объявлены в документах, прилагаемых к компиляторам. Комитет ANSI, подготавливая стандарт языка, включил большинство этих изменений, а также ввел другие значительные модификации. Некоторые коммерческие компиляторы реализовали их еще до выпуска официального Си-стандарта.

В этом приложении сведены воедино различия между языком, определенным в первой его редакции, и той его версии, которая принята в качестве стандарта. Здесь рассматривается только сам язык; вопросы, относящиеся к его окружению и библиотеке, не затрагиваются. Хотя последние и являются важной частью стандарта, но, поскольку в первом издании не делалось попытки описать среду и библиотеку, с соответствующими стандартными элементами сравнивать практически нечего.

- В стандарте более тщательно, по сравнению с первым изданием, определено и расширено препроцессирование: в его основу явно положены лексемы; введены новые операторы для "склеивания" лексем (`##`) и создания символьных строк (`#`), а также новые управляющие строки, такие как ttelif и `#pragma`; разрешено повторное определение макроса с той же последовательностью лексем; отменена подстановка параметров внутри строк. Разрешено "склеивание" строк с помощью знака \ в любом месте, не только в строках и макроопределениях (см. А12).

- Минимальное число значимых символов всех внутренних идентификаторов доведено до 31; для идентификаторов с внешней связью оно остается равным 6; буквы нижнего и верхнего регистров не различаются. (Многие реализации допускают большее число значимых символов.)

- Для знаков#, \, X, [,], {, }, !, D, которых может не быть в некоторых наборах символов, введены трехзнаковые последовательности, начинающиеся с ?? (см. А12.1). Следует заметить, что введение трехзнаковых последовательностей может повредить значения строк, в которых содержатся ??.

- Введены новые ключевые слова (void, const, volatile, signed, enum), а мертворожденное слово entry из обращения изъято.

- Для символьных констант и строковых литералов определены новые эскейп-последовательности. Объявлено, что появление за \ символов не из принятых эскейп-последовательностей приводит к непредсказуемому результату (см. А2.5.2.).

- Узаконено полюбившееся всем тривиальное изменение: 8 и 9 не являются восьмеричными цифрами.

- Введен расширенный набор суффиксов для явного указания типов констант: U и L – для целых и F и L - для типов с плавающей точкой. Уточнены также правила определения типа для констант без суффиксов (А2.5).

- Объявлено, что соседние строки конкатенируются.

- Предоставлены средства, позволяющие записывать строковые литералы и символьные константы из расширенного набора символов (А2.6).

- Объекты типа char (как и объекты другого типа) можно специфицировать явно со знаком или без знака. Исключается использование словосочетания long float в смысле double, но вводится тип long double для чисел с плавающей точкой повышенной точности.

- С некоторых пор доступен тип unsigned char. Стандарт вводит ключевое слово signed для явного указания, что объект типа char или другого целочисленного типа имеет знак.

- Уже несколько лет в большинстве реализаций доступен тип void. Стандарт вводит void * в качестве типа обобщенного указателя; раньше для

этой цели использовали char *. Одновременно вступают в силу правила, по которым запрещается без преобразования типа "смешивать" указатели и целые или указатели разных типов.

- Стандарт устанавливает минимальные пределы диапазонов арифметических типов, предусматривает заголовочные файлы <limits.h> и <float.h>, в которых помещаются эти характеристики для каждой конкретной реализации.

- Перечисление - новый тип, которого не было в первой редакции.

- Стандарт заимствует из Си++ способ записи квалификатора типа, в частности квалификатора const (А8.2).

- Вводится запрет на модификацию строк; это значит, что их разрешается размещать в памяти, доступной только на чтение (ПЗУ).

- Изменены "обычные арифметические преобразования"; по существу, выполнен переход от принципа "для целых всегда превалирует unsigned; для плавающей точки всегда используется double" к принципу "повышение до минимального достаточно вместительного типа" (см. А6.5).

- Отменены старые операторы присваивания вроде =+. Каждый оператор присваивания теперь представляется одной отдельной лексемой. В первом издании оператор присваивания мог изображаться парой символов, возможно, разделенных символами-разделителями.

- Компиляторам более не разрешается трактовать математическую ассоциативность операторов как вычислительную ассоциативность.

- Введен унарный оператор + для симметрии с унарным -.

- Разрешено использовать указатель на функцию в качестве ее именующего выражения без явного оператора * (см. А7.3.2).

- Структурами разрешено оперировать при присваиваниях, можно передавать структуры в качестве аргументов функциям и получать их в качестве результата от функций.

- Разрешено применять оператор получения адреса & к массиву; результатом является указатель на массив.

- В первой редакции результат операции sizeof имел тип int; во многих реализациях он заменен на unsigned. Стандарт официально объявляет его зависимым от реализации, но требует, чтобы он был определен в заголовочном файле <stddef.h> под именем size_t. Аналогичное изменение было сделано в отношении типа разности указателей, который теперь выступает под именем ptrdiff_t (см. А7.4.8 и А7.7).

- Запрещено применять оператор получения адреса & к объекту типа register даже тогда, когда данный компилятор не располагает его на регистре.

- Типом результата операции сдвига является тип ее левого операнда; тип правого операнда на повышение типа результата влияния не оказывает (см. А7.8).

- Стандарт разрешает адресоваться с помощью указателей на место, лежащее сразу за последним элементом массива, и позволяет оперировать с такими указателями, как с обычными, см. А7.7.

- Стандарт вводит (заимствованный из Си++) способ записи прототипа функции с включением в него типов параметров и явного указания о возможности их изменения и формализует метод работы с переменным списком аргументов. (См. А7.3.2, А8.6.3, В7.) С некоторыми ограничениями доступен и старый способ записи.

- Стандартом запрещены пустые объявления, т. е. такие, в которых нет объявителей и не объявляется ни одной структуры, объединения или перечисления. Однако объявление с одним тегом структуры (или объединения) переобъявит ее даже в том случае, если она была объявлена во внешней области действия.

- Запрещены объявления внешних данных, не имеющие спецификаторов и квалификаторов (т. е. объявления с одним "голым" объявителем).

- В некоторых реализациях, когда extern-объявление расположено во внутреннем блоке, его область видимости распространяется на остальную часть файла. Стандарт вносит ясность в эту ситуацию и объявляет, что область видимости такого объявления ограничена блоком.

- Область видимости параметров "вставляется" в составную инструкцию, представляющую собой тело функции, так что объявления на верхнем уровне функции не могут их "затенить".

- Несколько изменены именные пространства идентификаторов. Всем тегам структур, объединений и перечислений стандарт выделяет одно именное пространство; для меток инструкций вводится отдельное именное пространство (см. А11.1). Кроме того, имена элементов связаны со структурой или объединением, частью которого они являются. (С некоторых пор это общепринятая практика.)

- Допускается инициализация объединения; инициализатор относится к первому элементу объединения.

- Разрешается инициализация автоматических структур, объединений и массивов, хотя и с некоторыми ограничениями.

- Разрешается инициализация массива символов с помощью строкового литерала по точному количеству указанных символов (без '\0').

- Управляющее выражение и case-метки в switch могут иметь любой целочисленный тип.

Предметный указатель

А

Б

В

Г

Д

Е

З

И

К

П

Х

Ц

Ч

Ш

Э

A

B

C

D

E

F

T

U

V

W

X

Научное издание

Брайан В. Керниган, Деннис М. Ритчи

Язык программирования Си

Издательство "Невский Диалект"
195220, Санкт-Петербург, Гражданский пр., 14.
Лицензия на издательскую деятельность
серия ЛР № 065012 от 18.02.97.

Подписано в печать 29.11.2000. Формат 60×90⅟16
Бумага газетная. Печать офсетная.
Гарнитура *Petersburg C.*
Усл. печ. л. 22. Доп. тираж 4000 экз. Заказ № 1116

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-7940-0045-7

